TEMPOS: A Platform for Developing Temporal Applications on Top of Object DBMS

Marlon Dumas, *Member*, *IEEE Computer Society*, Marie-Christine Fauvet, and Pierre-Claude Scholl

Abstract—This paper presents TEMPOS: a set of models and languages supporting the manipulation of temporal data on top of object DBMS. The proposed models exploit object-oriented technology to meet some important, yet traditionally neglected design criteria related to legacy code migration and representation independence. Two complementary ways for accessing temporal data are offered: a query language and a visual browser. The query language, namely TEMPOQL, is an extension of OQL supporting the manipulation of histories regardless of their representations, through fully composable functional operators. The visual browser offers operators that facilitate several time-related interactive navigation tasks, such as studying a snapshot of a collection of objects at a given instant, or detecting and examining changes within temporal attributes and relationships. TEMPOS models and languages have been formalized both at the syntactical and the semantical level and have been implemented on top of an object DBMS. The suitability of the proposals with regard to applications' requirements has been validated through concrete case studies.

Index Terms—Temporal database, temporal data model, temporal query language, time representation, object database.

1 INTRODUCTION

TEMPORAL data handling is a pervasive aspect of many applications built on top of Database Management Systems (DBMS). Accordingly, most of these systems provide datatypes corresponding to the concepts of date and span. These datatypes are adequate for modeling simple temporal associations such as the date of birth or the age of a person. However, they are insufficient when it comes to model more complex temporal data, such as the history of an employee's responsibilities, or the sequence of annotations attached to a video. Since no datatypes dedicated to these kinds of associations are currently provided by DBMS, type constructors such as "list" and "tuple" must be used instead to encode them. The semantics of this encoding must then be integrated into the application programs, thereby increasing their complexity. Temporal database systems aim at overcoming these deficiencies [1], [2], [3], [4].

Research in this area has been quite prolific regarding extension proposals to data models and query languages. Whereas, in the relational framework, these works have led to the consensus language TSQL2 [5], there is no equivalent result in the object-oriented framework. Early attempts to define temporal extensions of object-oriented data models [2] had a limited impact, essentially due to the absence of a standard underlying data model. As the ODMG [6] proposal was released and started to be adopted by the major object DBMS vendors, a few temporal extensions of it were defined, among which TOOBIS [7] and T_ODMG [8]. However, we argue that these proposals lack at least some of the following four important features:

- Migration support, as to ensure a seamless transition of applications running on top of a nontemporal system to a temporal extension of it.
- Encapsulation of temporal types, as to separate the semantics of temporal data from its internal encoding.
- Formal semantics, to avoid many ambiguities generated by the richness and complexity of temporal concepts, and to serve as a basis for efficient implementation.
- Visual interfaces supporting user tasks such as navigating through a collection of temporal objects.

The goal of the TEMPOS project (Temporal Extension Models for Persistent Object Servers) [9], [10], [11], [12] has been to contribute toward a consensus view on how to handle temporality in object-oriented models by defining a temporal object database framework integrating the above features. This paper summarizes the results of this effort. The proposed framework is based on a temporal data model, on top of which two interfaces for retrieving and exploring temporal objects are provided: the TEMPOQL query language and the pointwise temporal object browser.

The paper is structured as follows: Section 2 focuses on defining the requirements related to application migration and representation independence and shows why existing temporal extensions of ODMG fail to fulfill them. Section 3, describes the TEMPOS data model, and Section 4 presents the query language and the visual browser. In Section 5, we present the prototype that has been developed to validate the feasibility of our proposal and we describe a major application among other that have been implemented on top of TEMPOS. Finally, in Section 6, we end with an overview of the proposal and point some future research directions.

- *M. Dumas is with the Centre for Information Technology Innovation, Queensland University of Technology, Brisbane Qld 4001, Australia. E-mail: m.dumas@qut.edu.au.*
- *M.-C. Fauvet and P.-C. Scholl are with the University of Grenoble, Lab. LSR - IMAG, BP 53, 38041 Grenoble Cedex 9 - France. E-mail: {marie-christine.fauvet, pierre-claude.scholl}@imag.fr.*

Manuscript received 10 Jan. 2001; revised 10 Dec. 2001; accepted 19 Nov. 2002.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 113443.

2 Motivation and Related Work

In this section, we present some of the major requirements that guided the design of TEMPOS. These requirements are divided into two categories: those which deal with the migration of data and application programs from a nontemporal to a temporal environment and those which deal with the abstract modeling and querying of histories. Before reviewing these requirements, we describe the application used throughout the article.

2.1 Application Example

We consider an application dealing with a factory's assembly lines and employees. Assembly lines are modeled by the class `AssemblyLine`. Each assembly line is identified by a number (attribute `lineCode`) and has a history of daily productions (attributes `production` and `quality`). At a given date, an assembly line has a set of workers who are assigned to it (attribute `workers`) and is under the responsibility of an employee (attribute `supervisor`). In addition, at a given date, each employee (modeled by the class `Employee`) identified by her(his) name (attribute `name`), earns a salary (attribute `salary`). The class `Employee` is specialized into two subclasses, `Supervisor` and `Worker`. The former contains the employees who supervise an assembly line, while the latter contains the employees who are allocated to an assembly line as workers. At a given date, each supervisor oversees a number of units (attribute `supervises`) and each worker is allocated to one unit (attribute `assemblyLine`).

For all attributes, except `lineCode` in `AssemblyLine` and `name` in `Employee`, the whole history of their values is stored. For example, the salary of an employee is a temporal attribute: All the past salaries of an employee are recorded in the database. Moreover, histories of facts are recorded with respect to valid-time. As defined in [13], the *valid time* of a fact is the time when the fact is true in the modeled reality (e.g., John's salary is 1,000 in *June 2000*). In contrast, the *transaction time* of a fact is the time when the fact is current in the database and can be retrieved (e.g., the fact that John's salary is 1,000 in June 2000 has been recorded on the *20th of May 2000*). A fact can also be bitemporal, meaning that it involves both valid and transaction times.

Finally, all the classes in the application are temporal, in the sense that both "currently valid" and "currently invalid" objects are recorded. This means that employees who have quit the company are recorded in addition to currently active employees.

2.2 Migration Requirements

Most of the temporal data models and languages that have been proposed in the literature are extensions of "conventional" ones. A common rationale for this design choice is that the resulting models can be integrated into existing systems, so that applications built on top of these systems may rapidly benefit from the added technology. However, the smooth migration of existing data and application programs to temporal database systems may only be achieved if these latter fulfill some elementary compatibility requirements. Such requirements have been formally defined in the context of temporal extensions to relational models by Bohlen et al. [14], [15]. However, the adaptation and application of these requirements to an Object-Oriented (OO) setting has not been previously considered. This adaptation is not completely trivial due to the following reasons:

- OO models rely on different concepts than relational models: objects, classes, types, properties, and methods instead of relations, tuples, and attributes.
- OO models provide a different (and more eclectic) set of update operators: object creation, destruction, dereferencing, attribute updating, and traversal path updating, instead of SQL's INSERT, DELETE, and UPDATE.
- OO databases support direct access from programming languages (e.g., from C++ and Java), in addition to database retrieval through a query language (e.g., OQL).

In the following, we consider two levels of migration requirements: *upward compatibility* and *temporal transitioning support*. These requirements are adaptations to the object-oriented setting of the concepts of upward compatibility and temporal upward compatibility defined in [15].

A data model D1 is said to be upward compatible with another data model D2, if every instance of D2 is also an instance of D1. In particular, a temporal data model is said to be upward compatible with a nontemporal data model, if it supports nontemporal database instances. To illustrate upward compatibility, consider an ODMG compliant DBMS managing a database about document loans in a library. Upward compatibility states that, if the ODMG DBMS is replaced by a temporal extension of it, the application programs accessing these data may be left intact. This implies that the set of database instances recognized by the extension is a super-set of those recognized by the original DBMS and that the query and update statements have identical semantics in the original DBMS and in the temporal extension.

Now, suppose that, once the legacy applications run on the temporal extension, it is decided that the history of the loans should be kept, but in such a way that legacy application programs may continue to run (at worst they should be recompiled), while new applications should perceive the property as being "historical." We call this requirement *temporal transition support*.

While upward compatibility can be achieved by simply adding new concepts and constructs to a model without modifying the existing ones, temporal transitioning support is more difficult to achieve. Bohlen et al. [15] point out that almost none of the existing temporal extensions to SQL, including TSQL2, satisfy this latter requirement. It can be shown that the same remark holds for existing object-oriented temporal extensions and, in particular, for the T_ODMG [8] temporal object model. For example, consider a `Document` class with a property `loaned_by` defined on it. In the context of T_ODMG, if some temporal support is attached to this property, then any subsequent access to it will retrieve not only the current value of the document's `loaned_by` property (as in the snapshot version of the database), but also its whole history. The same remark applies to the models defined by Goralwalla and Ozsu [16] and Rose and Segev [17].

TOOBIS [7] does not exhibit this migration problem. However, in achieving temporal transitioning support, TOOBIS introduces some burden to temporal applications. Indeed, in TOOBIS TOQL for example, each reference to a temporal property should be prefixed by either keyword `valid`, `transaction`, or `bitemporal` depending on the relevant time dimension(s). This leads to rather cumbersome query expressions. Similar remarks apply to TOOBIS C++ binding. This approach is actually equivalent to duplicating the symbols for accessing data when adding temporal support in such a way that, for each temporally enhanced property `p`, there are actually two properties representing it in the database schema: `p` and `valid p` (this latter is a temporal property). In the example of the library database, this means that when adding temporal support to `loaned_by`, this property is actually not modified and instead, a new temporal property is added, namely `valid loaned_by`.

We advocate a different approach. When temporal support is added to a database schema `S`, yielding a new schema `S'`, application programs are divided into two categories: those which view data as if its schema was `S` and those which view it with schema `S'`. Therefore, the problem of temporal transitioning support is seen as a form of schema evolution, so that techniques developed in this context apply. The reason for adopting this approach instead of TOOBIS's one may be stated simply: If a property is modified to add temporal support, temporal applications should perceive this property as being temporal.

Another migration requirement defined in [15] that has been taken into account in our proposal is that of *S-reducibility*. The idea is that all the operators defined by a nontemporal query language should be mirrored in the temporal extension of this language and given a temporal semantics when applied to temporal data. For example, the navigation operator defined by OQL should be given a temporal semantics when applied to temporal properties. This ensures that the semantics of temporal queries is understandable in terms of the semantics of nontemporal queries, so that a programmer familiar with the nontemporal query language can apply his/her knowledge to formulate temporal queries. This requirement is a particular case of the one introduced in the next section, which promotes the manipulation of temporal data through operators that abstract from the underlying representation.

2.3 Representation Independence

An elementary temporal association is a piece of data relating a fact to an instant. In most temporal data models, elementary temporal associations are grouped into temporal relations (in the relational framework) or into object or attribute histories (in the object and the object-relational frameworks). Temporal relations and histories can be represented in several ways. For example, it is possible to associate an instant to every tuple in a temporal relation at the logical level [18].[1] An alternative is to group several value-equivalent tuples into a single one, time stamped either by a temporal element as in [19], or by an interval [17], [7], [8]. This latter is by far the most common approach in the temporal database literature. In these data models, queries on temporal relations (or attribute histories) are defined over an interval time-stamped representation. This leads to some undesirable tensions between query expressions and their intended semantics. The following query expressed in TOOBIS TOQL query language illustrates this point.

When has the assembly line supervised by employee X had a quality-weighted production greater than that of the assembly line supervised by employee Y?

```
/*The extent of the class Supervisor is denoted
TheSupervisors.*/
select valid(pX) from TheSupervisors as supX,
TheSupervisors as supY,
valid supX.supervises as lineX,
valid supY.supervises as lineY,
valid line X.production as prodX,
valid lineY.production as prodY,
valid lineX.quality as qualX,
valid lineY.quality as qualY
where supX.name = "X" and supY.name = "Y" and
prodX * qualX > prodY * qualY and valid(lineX)
overlaps valid(lineY) and
valid(prodX) >= begin(valid(lineX)) and
valid(prodX) <= end(valid(lineX)) and
valid(prodY) >= begin(valid(lineY)) and
valid(prodY) <= end(valid(lineY)) and
valid(prodX) = valid(prodY) and
valid(qualX) = valid(qualY) and
valid(qualX) = valid(prodX)
```

This query expresses a join between the quality and the production histories of the assembly lines supervised by X and Y. The following time stamps and collections of time-stamped objects are involved in the query:

- `valid supX.supervises` is a collection of interval time-stamped `AssemblyLine` objects (the lines that X supervised).
- `valid(lineX)` is the validity period of a time-stamped assembly line, while `lineX` is the `AssemblyLine` object without the time stamp.
- `valid lineX.production` is a collection of instant time-stamped integers: the successive production volumes of the lines supervised by X.
- `valid lineX.quality` is a collection of instant time-stampted real numbers: the successive production qualities of the line supervised by X.
- `valid(prodX)` is the validity instant of a time-stamped integer denoting a production volume at a given date, while `prodX` is the value of this integer, without the time stamp.
- `valid(qualX)` is the validity instant of a time-stamped real number denoting a production quality at a given date, while `qualX` is the value of this real number, without the time stamp.

1. In [18], an instant time-stamped representation is adopted at the logical level, while an interval time-stamped representation is used at the physical level.

In this example, there is a clear difference in the level of abstraction between the query expression in natural language and that in TOQL. In the natural language formulation, there is no reference to any time-stamped object, whereas the TOQL formulation is fully based on time-stamp manipulation. This mismatch is the result of a lack of conceptuality in the language: TOQL lacks operators for reasoning about *simultaneity*. Specifically, no operators for navigating through temporal attributes and associations are provided. Instead, TOQL relies on an interval time-stamped representation of histories, together with operators over intervals for expressing most kinds of temporal queries. The same remark applies to many other proposals of temporal object query languages. For example, the expression of the above query in TIGUKAT [16] and TOOSQL [17] follows the same principle as that in TOOBIS: The `from` clause is used to declare a series of iterations over the histories involved in the query, while the `where` clause contains a series of join conditions between these histories. Hence, the navigation through temporal properties (or *behaviors* as termed in TIGUKAT) is diluted in the `from` and the `where` clauses. This gap between the conceptual expression of a temporal query and its implementation in TIGUKAT or TOOSQL also appears in other types of queries, such as those involving a succession of events in time, or a change of granularity. Indeed, to the best of our knowledge, there are no dedicated operators in TIGUKAT or TOOSQL for grouping a history according to a given granularity.

We advocate that exclusively relying on a fixed representation of temporal data to define the semantics of temporal operators, or for query expression, is an undesirable feature in a temporal object model. Instead, specific "representation independent" operators on histories should be provided, covering the fundamental temporal reasoning paradigms such as simultaneity, succession, and granularity change.

3 The TEMPOS Data Model

The TEMPOS data model is based on a set of datatypes whose behavior is described by type interfaces. The distinction between interfaces (abstract type descriptions) and classes (concrete implementations) is exploited to enforce the separation between the semantics of the operators over these datatypes and their implementation under some fixed representation. Furthermore, TEMPOS is structured into three increasingly sophisticated levels, i.e., the second level contains the first level, and the third level contains the second level. This structure enables a particular implementation to choose a degree of compliance according to the requirements of the targeted applications and the extensibility of the underlying DBMS.

The first level is composed of a set of datatypes modeling time values (instants, durations, intervals, and sets of instants), expressed at multiple granularities (time units). This level is sufficient for applications that involve simple temporal associations such as dated events (e.g., dates of birth, dates of appointment, etc.).

The second level introduces the concept of *history* and its associated operators. These operators can be used by application programs to filter, transform, and aggregate historical data. This level is appropriate for applications that need to manipulate complex temporal associations (e.g., time series) but do not require any specific support for performing updates according to valid time or transaction time. In particular, this level does not provide temporal transitioning support as defined in Section 2.2. Indeed, if an attribute is declared as being of type "History," then all the applications will perceive it as such.

The third level extends the concepts of class, attribute, and relationship as defined in the ODMG standard, leading to the concepts of *temporal class*, *temporal attribute*, and *temporal relationship*. This level supports valid time and transaction time. Update and retrieval operators over temporal classes and properties follow the semantics of these two time dimensions. In addition, this level provides temporal transitioning support.

3.1 Modeling Histories

TEMPOS is based on a discrete, linear, and bounded time model in which the time line is structured in a multi-granular way by means of *time units*. A time unit models a partition of the time line into disjoint and contiguous intervals (see [20] for a formal definition). Days, months, and years are examples of time units. Some time units are finer than others. For example, unit Day is finer than unit Month (written Day $\prec$ Month). Based on this temporal structure, the TEMPOS model defines four basic temporal datatypes: `Instant`, `Duration`, `Interval`, and `TSequence` (set of instants). A full description of these temporal datatypes and their operations is out of the scope of this paper (see [9]). Taking these datatypes as building blocks, the historical model presented in this section provides the foundation for capturing the evolution of data items over time.

3.1.1 Historical Model

A history is abstractly defined as a function from a finite set of instants to a set of values of a given type. The set of histories of a given type `T` are modeled through an abstract datatype `History`$\langle$`T`$\rangle$.

To describe the operators of the `History`$\langle$`T`$\rangle$ datatype, we use functional notations. Indeed, given that most of the operators on histories are higher-order operators (i.e., functions whose parameters are themselves functions), a simple UML-like description of them would not be accurate enough.

The following notations are used: `T1` $\rightarrow$ `T2` stands for the type of all functions with domain `T1` and range `T2`. `{T}` and `[T]`, respectively, denote the type of sets of `T` and sequences of `T`. $\langle$`T1, T2, ..., Tn`$\rangle$ designates the type of tuples whose ith component is of type `Ti` ($1 \leq i \leq n$); tuple components may be labeled using the notation $\langle$`L1 : T1, L2 : T2, ..., Ln : Tn`$\rangle$. $\langle$`v1, v2, ..., vn`$\rangle$ denotes a tuple value whose ith component is `vi` ($1 \leq i \leq n$). If `x` is an

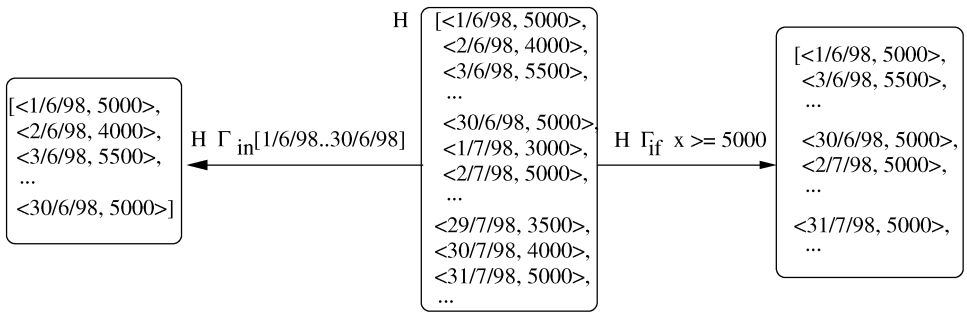

Fig. 1. Domain and range restrictions.

instant or a temporal sequence, then `Unit(x)` denotes its time unit.

The selectors of the `History`⟨`T`⟩ type are:

```
Domain: History⟨T⟩ → TSequence
   /* retrieves the domain of a history */
Range: History⟨T⟩ → { T }
   /* retrieves the range of a History */
Unit: History⟨T⟩ → Unit
   /* Unit(h) = time unit at which the history h
   is observed. */
Value: History⟨T⟩, Instant → T
   /* Value(H, I) is the value at instant I,
   assuming that I ∈ Domain(H) */
```

Histories may be represented in several ways, mainly by means of collections of time-stamped values, termed *chronicles*. Among these representations, some are useful for query expression, so that specific operators are defined, allowing one to convert a history into a chronicle. Concretely, a history may be represented by three kinds of chronicles:

- Chronologically ordered list of instant-timestamped values, e.g., `[`⟨`1, v1`⟩, ⟨`2, v1`⟩, ⟨`4, v1`⟩, ⟨`5, v2`⟩, ⟨`6, v2`⟩, ⟨`7, v2`⟩, ⟨`8, v3`⟩, ⟨`9, v1`⟩, ⟨`10, v1`⟩`]`. Such lists are termed `IChronicles`.
- Chronologically ordered, coalesced list of interval-time-stamped values, e.g., `[`⟨`[1..2], v1`⟩, ⟨`[4..4], v1`⟩, ⟨`[5..7], v2`⟩, ⟨`[8..8], v3`⟩, ⟨`[9..10], v1`⟩`]`. This kind of list is called an `XChronicle`.
- Set of distinct values time stamped by disjoint temporal sequences, e.g., { ⟨`{1, 2, 4, 9, 10}, v1`⟩, ⟨`{5, 6, 7}, v2`⟩, ⟨`{ 8 }, v3`⟩ }.

These representations are sometimes useful for query expression, so that specific operators are defined that cast either of these representations into histories and vice-versa [21]. Here, we only describe the operator allowing to cast histories into `XChronicles`.

```
XChronicle: History⟨T⟩ →
 [⟨timestamp: Instant, value: T⟩]
 /* XChronicle (h) = [C1, ..., Cn] ⇒ ∀k ∈ [1..n − 1],
 Ck.timestamp < Ck+1.timestamp ∧
 (Ck.timestamp meets Ck+1.timestamp ⇒ Ck.value
 ≠ Ck+1.value) */
```

3.1.2 Algebraic Operators on Histories

Algebraic operators on histories are classified into two categories: intrapoint and interpoint. An operator is said to be *intrapoint* if the value of the resulting history at a given instant depends exclusively on the value of the argument histories at that instant, otherwise, it is said to be *interpoint*. This classification is closed under composition: The composition of two intrapoint operators yields an intrapoint operator and, similarly, for interpoint operators.

Intrapoint operators. Intrapoint operators are essentially generalizations to histories of the selection, join, and projection operators of the relational algebra. Five Intrapoint operators are defined in TEMPOS: two corresponding to the selection (also called restriction), two corresponding to the join, and one corresponding to the projection.

The operators Γ_{if} and Γ_{in} restrict the domain of a history to those instants at which a given condition is true. Their semantics is illustrated in Fig. 1.

```
_Γif_: History⟨T⟩, (T → boolean) → History⟨T⟩
        /* h Γif P = {⟨I, v⟩ | ⟨I, v⟩ ∈ h ∧ P(v)} */
_Γin_: History⟨T⟩, TSequence → History⟨T⟩
        /* h Γin S = {⟨I, v⟩ | ⟨I, v⟩ ∈ h ∧ I ∈ S} */
```

The temporal join allows to combine histories. Since two histories may have different domains, we distinguish the *inner* temporal join ($*_\cap$) from the *outer* one ($*_\cup$), depending on whether the resulting history's temporal domain is the intersection or the union of the domains of the arguments: `h1` $*_\cap$ `h2` is a history whose values are pairs obtained by combining "synchronous" values of `h1` and `h2` (i.e., values attached to the same instant); `h1` $*_\cup$ `h2` is similar, except that it attaches values of the form ⟨`v, nil`⟩ or ⟨`nil, v`⟩, to those instants where one of the argument histories is defined while the other is not. More precisely:

```
_*∩_: History⟨T1⟩, History⟨T2⟩ →
 History⟨⟨T1,T2⟩⟩
 /* h1 *∩ h2 = {⟨I, ⟨v1, v2⟩⟩ | ⟨I, v1⟩ ∈ h1 ∧ ⟨I, v2⟩ ∈ h2} */
 /* precondition: Unit(h1) = Unit(h2)} */
_*∪_: History⟨T1⟩, History⟨T2⟩ →
 History⟨⟨T1,T2⟩⟩
 /* h1 *∪ h2 = h1 *∪ h2 ∪ {⟨I, ⟨v1, nil⟩⟩ | ⟨I, v1⟩ ∈ h1 ∧ I ∉
 Domain(h2)}∪{⟨I, ⟨nil, v2⟩⟩ | ⟨I, v2⟩ ∈ h2 ∧ I ∉ Domain(h1)}
 precondition: Unit(h1) = Unit(h2)} */
```

The semantics of the join operators are informally illustrated in Fig. 2.

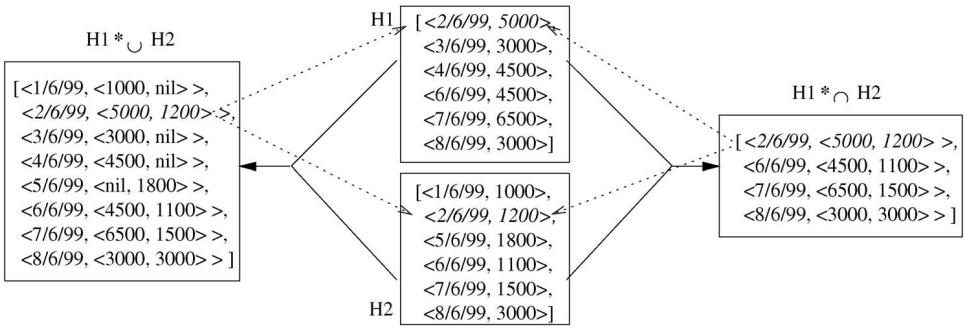

Fig. 2. Inner and outer joins.

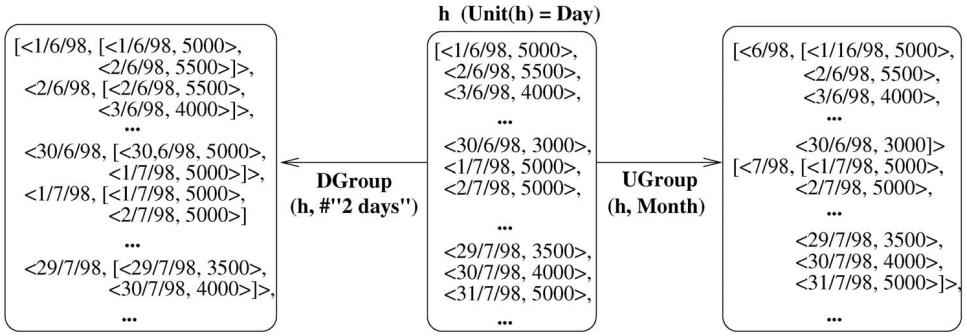

Fig. 3. Unit-based and duration-based temporal grouping.

The last of the Intrapoint operators, namely `Map`, applies a given function to each value of a history. For example, given a history `H` of integers, the operation `Map(h1, x / 100)` yields another history of integers, obtained by dividing the values of `H` by 100.

$$\texttt{Map: History}\langle\texttt{T}\rangle\texttt{, (T} \rightarrow \texttt{T')} \rightarrow \texttt{History}\langle\texttt{T'}\rangle$$
$$\texttt{*/ Map(h, f)} = \{\langle \texttt{I}, \texttt{f(v)}\rangle \mid \langle \texttt{I}, \texttt{v}\rangle \in \texttt{h}\} \texttt{ /*}$$

Interpoint operators. Interpoint operators include partitioning (or grouping) operators and operators dealing with succession in time. There are two grouping operators in TEMPOS: `UGroup` and `DGroup`. `UGroup (h, u2)`, `h` being at granularity `u1` (`u1` $\prec$ `u2`), divides up history `h` into groups according to unit `u2`. The result is a history at granularity `u2` of histories at granularity `u1` whose values at instant `i` are the temporal restriction of `h` to the interval `expand (i, u1)`.[2] This is illustrated in Fig. 3.

On the other hand, `DGroup (h, d)` yields all subhistories of `h` of duration `d`. The resulting history associates to instant `i` the restriction of `h` to interval $[\texttt{i..i} + \texttt{d}]$, if `d` is positive, or to $[\texttt{i} + \texttt{d..i}]$ if `d` is negative, provided that the corresponding interval is included in the temporal domain of `h`. This operator is useful to express moving window queries, as in *"compute the average sales for each seven-day period in the history of daily sales of a store."* Formally,

$$\texttt{UGroup: History}\langle\texttt{T}\rangle\texttt{, Unit} \rightarrow \texttt{History}\langle\texttt{History}\langle\texttt{T}\rangle\rangle$$
$$\texttt{*/ UGroup(h, u)} = \{\langle \texttt{I}, \texttt{subh}\rangle \mid \exists \texttt{I'} \in \texttt{Domain(h)},$$
$$\texttt{approx(I', u)} = \texttt{I} \wedge \texttt{h}\ \Gamma_{\texttt{in}}\ \texttt{expand(I, Unit(h))} = \texttt{subh}\} \texttt{ /*}$$
$$\texttt{*/ precondition : Unit(h)} \prec \texttt{u /*}$$
$$\texttt{DGroup : History}\langle\texttt{T}\rangle\texttt{, Duration} \rightarrow$$
$$\texttt{History}\langle\texttt{History}\langle\texttt{T}\rangle\rangle$$
$$\texttt{*/ DGroup(h, d)} =$$

$$\begin{cases} \{\langle \texttt{I}, \texttt{subh}\rangle \mid [\texttt{I..I} + \texttt{d}] \subseteq \texttt{h} \wedge \texttt{subh} = \texttt{h}\Gamma_{\texttt{in}}[\texttt{I..I} + \texttt{d}]\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{if d positive} \\ \{\langle \texttt{I}, \texttt{subh}\rangle \mid [\texttt{I..I} + \texttt{d}] \subseteq \texttt{h} \wedge \texttt{subh} = \texttt{h}\Gamma_{\texttt{in}}[\texttt{I} + \texttt{d..I}]\} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{if d negative} \end{cases}$$

$$\texttt{/* preconditions: Unit(d) = Unit(h) or Unit(d)}$$
$$\texttt{and Unit(h) are regular */}$$

To reason about successive values of histories and their correlations, TEMPOS provides four operators, namely, `AfterFirst`, `BeforeFirst`, `AfterLast`, and `BeforeLast`. These operators are algebraic versions of the "since" and "until" operators of linear temporal logics [22]. Specifically, `AfterFirst(h, P)` yields the subhistory of `h` starting at the first instant at which the value of `h` satisfies predicate `P`, or the empty history if such instant does not exist. `BeforeFirst(h, P)`, on the other hand, restricts `h` to those instants preceding the first instant at which the value of `h` satisfies `P`, or `h` if such instant does not exist. For any history `h` and any predicate `P`, `h` is equal to the union of `BeforeFirst(h, P)` and `AfterFirst(h, P)` (which are disjoint). Similar remarks apply to `AfterLast` and `BeforeLast` which are defined symmetrically.

$$\texttt{BeforeFirst: History}\langle\texttt{T}\rangle\texttt{,}$$
$$\texttt{(T} \rightarrow \texttt{boolean)} \rightarrow \texttt{History}\langle\texttt{T}\rangle$$
$$\texttt{/* BeforeFirst(h, P)} = \{\langle \texttt{I}, \texttt{v}\rangle \mid \langle \texttt{I}, \texttt{v}\rangle \in \texttt{h} \wedge \neg\exists\langle \texttt{I'}, \texttt{v'}\rangle \in$$
$$\texttt{h (P(v')} \wedge \texttt{I} \geq \texttt{I')}\} \texttt{ */}$$
$$\texttt{AfterFirst: History}\langle\texttt{T}\rangle\texttt{,}$$
$$\texttt{(T} \rightarrow \texttt{boolean)} \rightarrow \texttt{History}\langle\texttt{T}\rangle$$

2. Operation `expand (i, u)` maps an instant `i` observed at some unit `u2`, to an interval at a finer unit `u1`. For example, expand('January 2001') = ['1 January 2001'..'31 January 2001'].

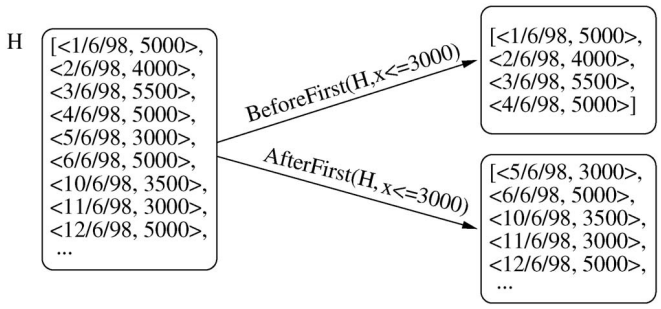

Fig. 4. History splitting operators.

/* AfterFirst(h, P) = $\{\langle I, v\rangle \mid \langle I, v\rangle \in h \wedge \exists \langle I', v'\rangle \in h\ (P(v') \wedge I \geq I')\}$ */

Fig. 4 provides an informal view of the semantics of `AfterFirst` and `BeforeFirst`.

3.2 Temporal Properties and Classes

In this section, we extend the concepts of class, property, and their instances to integrate temporal support. We also show how this extension fulfills the requirements formulated in Section 2.

3.2.1 Temporality at the Property Level

As in ODMG, a property is defined as an attribute or traversal path of a relation attached to some class. For instance, possible properties of an `Employee` class include `salary` and `department`. Instances of properties are attached to objects, so that *property instances* are to objects what *properties* are to classes.

The following two paragraphs successively describe: 1) how temporal support is attached to properties and 2) what is the effect of attaching such support on the values taken by property instances?

Temporal properties. In TEMPOS, a property may be either *temporal*, in which case its successive values are meaningful and thus recorded, or *fleeting*, if only its most recent value is meaningful. When a property is temporal, the granularity at which its evolution is observed is determined by a specific characteristic of the property, namely, its *observation unit*.

As in ODMG, a type is attached to a property. In the case of a fleeting property, this type defines the domain of possible values that an instance of this property may take, whereas for a temporal property, it models the values that an instance of this property may take at some instant. If the type of a temporal property is `T`, each of its instances has a history whose type is `History`$\langle$`T`$\rangle$.

The *temporal dimension* of a temporal property determines the semantics of the temporal associations that it models. It may be *valid-time* or *transaction-time* depending on whether the facts are timestamped with respect to the modeled reality or with respect to the database evolution [23]. TEMPOS therefore distinguishes *valid-time* and *transaction-time* properties. By merging these two concepts, it is possible to model bitemporal associations, although we do not address this issue in this paper.

Instances of temporal properties. The *(observation) domain* of an instance of a temporal property is the set of instants (i.e., the `TSequence`) during which the property is observed for a given object. Observation domains may evolve dynamically according to the system clock.

The *(full) history* of a temporal property instance is a history reflecting the values taken by the property instance at all instants when it is observed. The domain of this history is equal to the observation domain of the property instance. Its structural values, on the other hand, are either defined by some update, or derived from the inputted values using a *semantic assumption*. Specifically, a temporal property instance has an *effective history*, corresponding to the inputted time-stamped values attached to it. The effective history is contained in (but not necessarily equal to) the property instance's history, and the difference between them is called the *potential history*. This is the part of the full history derived through the semantic assumption. In the sequel, we will refer to the domain of a property instance's effective history as the *effective domain*.

We distinguish three semantic assumptions depending on the intended calculation mode of the potential history (see Fig. 5).

- *Discrete*: The structural value of the potential history is equal to the neutral value of its structural type (e.g., `0` for integers, `nil` for objects). This is the case of the production of a product in a factory: The period of time during which the production is defined (i.e., its observation domain) may be known in advance (e.g., all weekdays), but at some days, it may be that there is no inputted value (e.g., due to a strike), so that the effective history is not defined for those days. A *padding value* may be attached to a discrete property, to override the use of the neutral element of the property's type as the "default" value for the potential histories.
- *Stepwise*: Values are "stable" between two instants in the effective temporal domain (e.g., a property instance modeling an employee's salary). A padding value determines the value of the property instances at those time instants for which the stepwise semantic assumption does not provide one (e.g., when the smallest instant in the effective domain is not equal to the smallest instant in the observation domain).
- *Linearly interpolated* (only for numerically-valued properties): Between two successive instants in the effective domain, the value varies linearly.

Transaction-time properties have a stepwise semantic assumption. In addition, the temporal domains of their instances may evolve with the system clock. Each time that a transaction-time property instance is accessed, its temporal domain is computed by replacing its upper bound with the current instant, unless the property is "turned off" as discussed later. This is depicted in Fig. 6.

Temporal binary relationships. In ODMG, a *relationship* is defined implicitly through the declaration of a pair of inverse properties attached to the class(es) participating in the relationship. An association between a class `C1` and a class `C2` is modeled by attaching a property `P1` to `C1`, and a property `P2` to class `C2`. Depending on the multiplicity of the association, `P1` and `P2` are single-valued or set-valued. For example, if the multiplicity of the link going from `C1` to `C2` is "many," then `P1` is of type set $\langle$C2$\rangle$. Otherwise, if the multiplicity of this link is "one," then the type of `P1` is

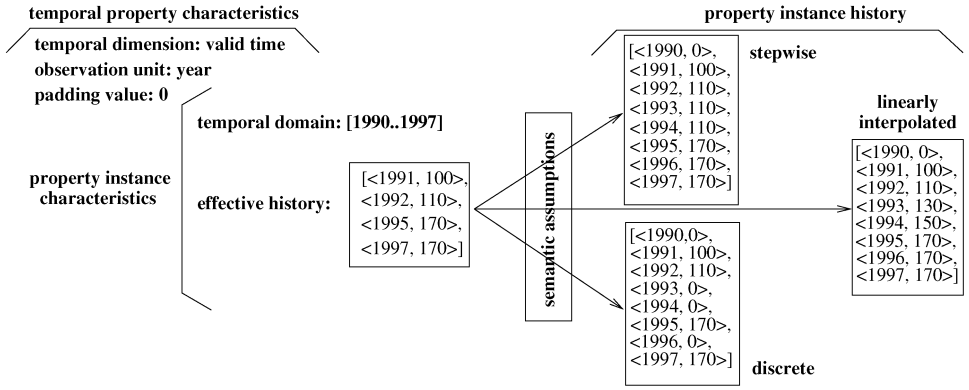

Fig. 5. A valid-time property instance's history is calculated based on the effective history and the semantic assumption.

simply C2. The DBMS automatically ensures referential integrity: If an object o1 of class C1 refers, through property P1, to an object o2 of class C2, then o2 refers to o1 through property P2.

TEMPOS extends this notion of inverse properties to model *temporal relationships*. More precisely, if P1 and P2, respectively, attached to classes C1 and C2 are the inverse properties defining a temporal relationship, then the constraints listed below hold. The following notations are used: extent(C) is the set of all instances of class C, and o.P is the history of property instance P attached to object o.

- If P1 and P2 are both single-valued then:

$$(\forall\ \mathtt{o1} \in \mathtt{extent(C1)}, \forall\ \mathtt{i} \in \mathtt{Domain(o1.P1)}\ \mathtt{i}\ \in \mathtt{Domain(Value(o1.P1, i).P2)} \wedge \mathtt{o1} = \mathtt{Value(Value(o1.P1, i).P2, i)}) \wedge (\forall\ \mathtt{o2} \in \mathtt{extent(C2)}, \forall\ \mathtt{i} \in \mathtt{Domain(o2.P2)}\ \mathtt{i} \in \mathtt{Domain(Value(o2.P2, i).P1)} \wedge \mathtt{o2} = \mathtt{Value(Value(o2.P2, i).P1, i)}).$$

- If P1 is single-valued and P2 is multivalued then:

$$(\forall\ \mathtt{o1} \in \mathtt{extent(C1)}, \forall\ \mathtt{i} \in \mathtt{Domain(o1.P1)}\ \mathtt{i} \in \mathtt{Domain(Value(o1.P1, i).P2)}) \wedge \mathtt{o1} \in \mathtt{Value(Value(o1.P1, i).P2, i)} \wedge (\forall\ \mathtt{o2} \in \mathtt{extent(C2)}, \forall\ \mathtt{i} \in \mathtt{Domain(o2.P2)}, \forall\ \mathtt{o1} \in \mathtt{Value(o2.P2, i)}\ \mathtt{i} \in \mathtt{Domain(o1.P1)} \wedge \mathtt{o2} = \mathtt{Value(o1.P1, i)}).$$

- If P1 and P2 are both multivalued then:

$$(\forall\ \mathtt{o1} \in \mathtt{extent(C1)}, \forall\ \mathtt{i} \in \mathtt{Domain(o1.P1)}, \forall\ \mathtt{o2} \in \mathtt{Value(o1.P1, i)}\ \mathtt{i} \in \mathtt{Domain(o2.P2)} \wedge \mathtt{o1} \in \mathtt{Value(o2.P2, i)}) \wedge (\forall\ \mathtt{o2} \in \mathtt{extent(C2)}, \forall\ \mathtt{i} \in \mathtt{Domain(o2.P2)}, \forall\ \mathtt{o1} \in \mathtt{Value(o2.P2, i)}\ \mathtt{i} \in \mathtt{Domain(o1.P1)} \wedge\ \mathtt{o2} \in \mathtt{Value(o1.P1, i)}).$$

3.2.2 Temporality at the Class Level

As properties, classes may be fleeting or temporal. A temporal class keeps track of its extent by associating to each of its instances the set of instants at which it is *observed*, either with respect to valid or transaction time. Instances of temporal classes are called *temporal objects*.

The *extent* of a class (whether fleeting or temporal) is defined as the set of all instances of this class having been created and not deleted. Due to the "append-only" semantics of transaction-time (i.e., no information may be lost), an object of a transaction-time class may not be deleted from its extent. The same applies to any object participating in a transaction-time relationship or referenced by a transaction-time attribute. For this reason, the operator delete is overloaded when applied to transaction-time objects as discussed later.

In addition to the notion of extent, two other notions are introduced that apply to temporal classes and their instances: observation domain and observed extent.

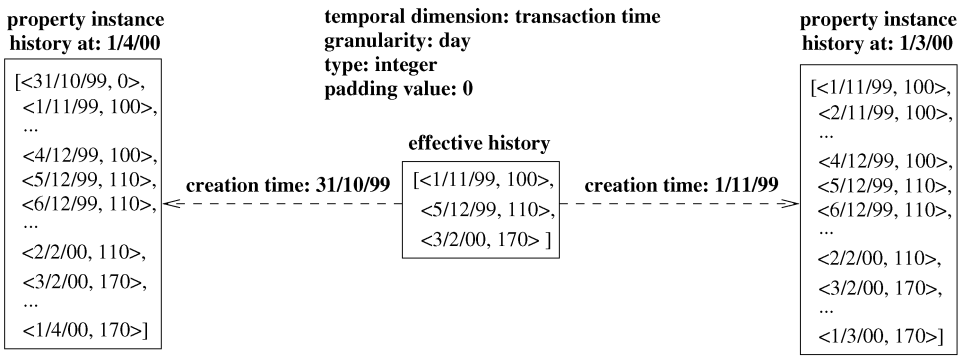

Fig. 6. The history of a transaction-time property instance is calculated based on the effective history, the creation time, and the current time.

Definition 1: Observation Domain. *A valid-time (respectively, transaction-time) object has a valid-time (respectively, transaction-time)* observation domain *attached to it, which is an arbitrary set of instants.*

Definition 2: Observed extent. *Given a temporal class, the* observed extent *of this class at an instant* `i` *is the subset of the extent consisting of all objects whose observation domain contains* `i`.

Conceptually, the observation domain of a valid-time or transaction-time object is the set of instants at which the information conveyed by this object is observed. The notion of "observation" may either be defined with respect to transaction-time (When is the information conveyed by an object observed in the database?), or to valid-time (When is the information about the entity modeled by an object observed?).

For example, consider a class `Product` modeling the product types produced and sold by a company. If the class is declared as temporal (either with respect to valid-time or transaction-time), then the observation domain could be used to model the time when a particular product is manufactured, or the time when it is sold. Suppose now that the observation domain models the time when the product is manufactured. If the class is transaction-time, then the value of the observation domain of an object of this class captures the time when the database knows that a product is manufactured, whereas, if the class is valid-time, it models the time when the corresponding product is actually manufactured in reality.

Temporal support on properties and class extents are orthogonal: A fleeting class may have transaction or valid-time properties and, reciprocally, a valid-time class or transaction-time class may have fleeting properties.

3.2.3 Updating and Accessing Temporal Property Instances

In ODMG, there is one "access" and one "update" operator for property instances, respectively, `get_value`, which retrieves the value of the property instance and `set_value` which assigns to it the value given as parameter. In the context of temporal property instances, the set of update and access operators is richer, due to the variety of temporal characteristics attached to them and the need to achieve biaccessibility (see Section 2). These operators are classified depending on whether they are intended to modify the observation domain or the effective history and depending on the time dimension (valid-time or transaction-time) to which they apply.

Evolution of the Observation Domain of Transaction-Time Property Instances. Since transaction-time is intended to model the evolution of the database, the observation domain of transaction-time properties instances evolves automatically with the system clock. Conceptually, the current instant is added to the observation domain of a transaction-time property instance at each clock tick. This automatic evolution of the observation domain can be overridden, so as to capture the fact that the property instance is not observed during some period of time. This is achieved through the notion of *growth status*, which takes one of two values: `On` or `Off`. If the value of the growth status of a transaction-time property instance is `On`, its observation domain evolves with the system clock. Otherwise, it does not evolve at all. Operators `turn_on` and `turn_off` on transaction-time property instances allow one to switch between these two states.

Evolution of the Observation Domain of Valid-Time Property Instances. Unlike transaction-time properties, the observation domain of a valid-time property instance does not evolve automatically with the system clock. Instead, an update operator `set_odomain` is provided, which destructively replaces the domain of the property instance by the `TSequence` given as parameter. Given that the observation domain must always contain the effective domain, this operator may force some modifications on the effective history, i.e., if the constraint is violated after some update to the observation domain, the effective history is restricted to fit inside this domain.

Evolution of the Effective History of Transaction-Time Property Instances. In the case of transaction-time properties, the effective history of a temporal property instance may only be modified by an overloaded version of ODMG's `set_value` operator. More precisely, `set_value(v)` applied to a transaction-time property instance `TTPI` replaces the effective history of `TTPI` by a new history, identical to the old one except that it maps the current instant to value `v`. If necessary, the growth status of the property instance is turned on.

Evolution of the Effective History of Valid-Time Property Instances. The operator `set_effective_history` destructively replaces the effective history of a temporal property instance with the one given as parameter. In order to achieve temporal transitioning support, the standard `set_value` operator is also supported and is given the following semantics (`VTPI` is a valid-time property instance):

$$\begin{aligned}\texttt{VTPI.set\_value(v)} \equiv\ & \texttt{VTPI.set\_effective\_history}\\ & (\texttt{VTPI.get\_effective\_history}()\\ & \triangleleft \{\langle\ \texttt{current\_instant}(), \texttt{v}\ \rangle\}).\end{aligned}$$

Where the operator $\triangleleft$ is defined as follows:

$$\begin{aligned}&\_ \triangleleft \_ : \texttt{History}\langle\texttt{T}\rangle, \texttt{History}\langle\texttt{T}\rangle \rightarrow \texttt{History}\langle\texttt{T}\rangle\\ &/^{*}\ \texttt{h1} \triangleleft \texttt{h2} = \{\langle \texttt{I}, \texttt{v}\rangle \mid \langle \texttt{I}, \texttt{v}\rangle \in \texttt{h2} \vee (\langle \texttt{I}, \texttt{v}\rangle\\ &\qquad \in \texttt{h1} \wedge \texttt{I} \notin \texttt{Domain(h2)})\}.\ /^{*}\end{aligned}$$

Accessing Temporal Property Instances. The basic operators to access temporal property instances are `get_effective_history` and `get_history`. The former simply retrieves the effective history of the temporal property instance, while the latter builds a history from the observation domain and the effective history using the corresponding semantic assumption as depicted in Figs. 5 and 6. To achieve temporal transitioning support, the ODMG `get_value` access operator is also supported on temporal property instances. Its semantics in this context is defined as follows (`TPI` is a temporal property instance):

$$\begin{aligned}\texttt{TPI.get\_value()} \equiv\ & \texttt{Value(TPI.get\_history(),}\\ & \texttt{current\_instant()).}\end{aligned}$$

TABLE 1
Valid-Time Update Scenario

Scenario	A product is introduced on 1/4/98 with price 50
Operation	P = new Product; P.set_odomain([1/4/98..]);
	P.price.set_effective_history({⟨1/4/98, 50⟩})
Result	P.price.get_history() = {⟨[1/4/98..], 50⟩}
Scenario	On 1/6/98, the product's price raises to 60
Operation	P.price.set_effective_history(P.price.get_effective_history() $\cup_+$ {⟨1/6/98, 60⟩})
Result	P.price.get_history() = {⟨[1/4/98..31/5/98], 50⟩, ⟨[1/6/98..], 60⟩}
Scenario	The product is not longer produced starting from 6/8/98
Operation	P.set_odomain(P.get_odomain() ∩ [..5/8/98])
Result	P.get_odomain() = [1/4/98..5/8/98]
	P.price.get_history() = {⟨[1/4/98..31/5/98], 50⟩, ⟨[1/6/98..5/8/98], 60⟩}
Scenario	The product is produced again starting from 1/1/99; its price is not observed then
Operation	P.set_odomain(get_odomain(P) ∪ [1/1/99..])
Result	P.get_odomain() = {[1/4/98..5/8/98], [1/1/99..]}
	P.price.get_history() unchanged
Scenario	The product's price is observed starting from 1/2/99 and until 31/3/99. Its value at 1/2/99 is 80
Operation	P.price.set_odomain(P.price.get_odomain() ∪ [1/2/99..31/3/99])
	P.price.set_effective_history(P.price.get_effective_history() $\cup_+$ {⟨1/2/99, 80⟩})
Result	P.get_odomain() unchanged
	P.price.get_history() = {⟨[1/4/98..31/5/98], 50⟩, ⟨[1/6/98..5/8/98], 60⟩, ⟨[1/2/99..31/3/99], 80⟩}

3.2.4 Temporal Objects' Observation Domain Evolution

Evolution of Transaction-Time Observation Domains. The observation domain of transaction-time objects evolves automatically with the system clock in a similar way as the observation domains of transaction-time property instances. Each transaction-time object has a growth status, which may be `On` or `Off`. Conceptually, while a transaction-time object is on, the current instant is added to its observation domain at each clock tick. When a transaction-time object is created, its growth status is on. If the operator `delete` is called on it, the growth status turns off, but the object remains in the extent of its class. Subsequently, the growth status can be turned on again by calling the operator `revive`. There is no operator for suppressing a transaction-time object from its class extent. This is in line with the append-only semantics of transaction time [13].

Evolution of valid-time observation domains. When a valid-time object is created, its observation domain is set to be the interval `[current_instant()..]`, that is the interval starting at the current instant and extending up to the largest instant recognised by the system. This observation domain can be subsequently modified using operator `set_odomain`. This operator destructively sets the observation domain of the object to be the temporal sequence given as parameter. The operator `delete`, when applied on valid-time objects, does not physically delete the object from its extent. Instead, it sets the upper bound of the object's observation domain, to be the current instant. That is: `O.delete()` $\equiv$ `O.set_odomain(O.get_odomain()` $\cap$ `[..current_instant()]`, where `get_odomain` is an operator which returns the observation domain of an object, whether valid-time or transaction-time. The above definition of the operator `delete` on valid-time objects is crucial for ensuring temporal transitioning support as discussed in Section 3.3.2.

To enable physical deletion of a valid-time objects, an operator `destroy` is provided. This operator removes a valid-time object from its class extent, as does the operator `delete` when applied to nontemporal objects. The operator `destroy` raises an exception if the object to which it is applied participates in a transaction-time relationship since data items time stamped with transaction-time are indelible.

3.2.5 Example

We consider a class `Product` with a valid-time stepwise attribute `price` of type `real`. The valid-time observation domain of an object of class `Product` models the days when the product is manufactured, while the observation domain of an instance of property `price` models the time when its price is defined. Table 1 illustrates a possible update scenario.

3.3 Achieving Temporal Transitioning

After a database schema is modified to add temporal support to some of its classes and/or properties, applications may either continue to access the database as if there was no temporal component in the schema, or take into account the schema modification. The following paragraphs describe how objects are adapted after this kind of schema modification and how the temporal and nontemporal "views" of the database may be accessed by applications.

3.3.1 Database Instance Adaptation

To specify how a database instance is adapted to a schema modification which adds temporal support to nontemporal schema components, an object conversion operator is defined. Conceptually, this conversion operator should be applied to all existing objects in the database instance whenever the schema is modified. The following algorithm implements the object conversion.

Algorithm 1: *Object conversion operator*.

```
modified_properties: set of properties;
/* properties to which temporal support is
```

```
added */
modified_classes: set of classes;
/* classes to which temporal support is added */
o: object; /* object to be converted */
copy_of_o: object; /* used to store a copy of o */
copy_of_o := o.copy(); /* makes a copy of o */
if classOf(o) in modified_classes then
 if (temporalDimension(o) = transaction-time)
 then o.turn_on();
 else o.set_odomain([current_instant()..])
for p in (properties(classOf(o))
 if (p in modified_properties) then {
    if (temporalDimension(p) =
  transaction-time)
  then o.p.turn_on();
  else  if (semanticAssumption(p) = stepwise)
       then o.p.set_observation_domain
            ([current_instant()..])
  o.p.set_value(copy_of_o.p.get_value())
 } else o.p = copy_of_p.p
 /* property remains fleeting */
```

As shown by this algorithm, temporal migration support is only ensured for transaction-time and valid-time stepwise properties. This is because the idea behind temporal migration support is that when the "current" value of a temporal property is modified, the new current value assigned to it should remain constant. Such a characteristic is proper to stepwise properties.

3.3.2 Access Modes

Object-oriented programming and query languages generally only provide one construct for accessing property values and one for updating them. For instance, in C++, the only way of accessing the value of an attribute is through the "dot" operator (i.e., the operator `o.p`), whereas updating is performed through constructs of the form `o.p = v`. TEMPOS, on the other hand, provides several update and access primitives for each type of temporal property instance. The notion of *access mode* establishes which update or access operator on temporal properties is to be used depending on the application context. Two access modes are provided:

- The upward compatible mode: Temporal property instances are snapshot-valued: Their value is the value of their history at the current instant. In addition, any reference to the extent name of a temporal class (whether valid-time or transaction-time) retrieves the observed extent at the current instant.
- The temporal mode: Temporal property instances are history-valued, and no filtering is performed when accessing a temporal class extent (i.e., all objects in the extent are retrieved).

In the upward compatible mode, whenever a temporal property is accessed either from a program or from a query, the value associated to this property is retrieved through the `get_value` operator (see Section 3.2.3). Similarly, updates in this mode are handled by the `set_value` operator. In the temporal mode, `get_history` and `set_effective_history` are used instead.

If the operator `delete` is applied over a temporal object, this object is still visible by the temporal applications since it is still present in the extent of its class. However, it is not visible by the applications which are in upward compatible mode since its observation domain does not contain the current instant and, hence, it does not appear in the observed extent at the current instant (see Section 3.2.4).

To illustrate the role of access modes from the querying viewpoint, consider a class `Product` whose extent is named `TheProducts` and having two attributes `trademark` and `price` with types *string* and *real*, respectively. Suppose that, at some time during the life of the application, the class `Product` as well as its attribute `price` are declared as temporal (`trademark` remains a fleeting property).

Now, consider the following OQL query: `select struct(t : b.trademark, p : b.price) from TheProducts as b`. In the temporal mode, this query has type `bag⟨struct⟨t: string, p: History⟨real⟩⟩⟩` and retrieves, for each product type ever sold, its trademark and the history of its prices. Conversely, if the snapshot mode is assumed, the query has type `bag⟨struct⟨t: string, p: real⟩⟩` and retrieves the trademarks of all currently sold product types with their prices.

Different access modes may be attached to any two applications accessing the same database. As a result, the access mode should be implemented on a particular system as a parameter of each application session. The upward-compatible mode is the default in TEMPOS. This design choice ensures temporal transition support.

4 Querying and Browsing

4.1 Querying Temporal Objects

TEMPOQL extends OQL with types such as time unit, instant, interval, and history together with language constructs for manipulating the values of these types. Below, we introduce some of the salient constructs on histories provided by TEMPOQL, and illustrate them with examples taken from the application presented above. We assume that the application that submits these queries is in the temporal mode.

4.1.1 Formalization of TEMPOQL

TEMPOQL's constructs are formalized using a notation similar to that of [24], which provides a complete formalization of OQL. The formalization of a construct is made up of four parts:

- A *context* describing constraints on the types appearing in the typing part and on the subqueries (such as a variable being free in a subquery). Some of the typing preconditions refer to subtypes of `History` and `Instant`, even though none of these subtypes are defined by the model. This is to achieve extensibility, by allowing TEMPOQL constructs to be applicable to user-defined subtypes of `History` and `Instant`.

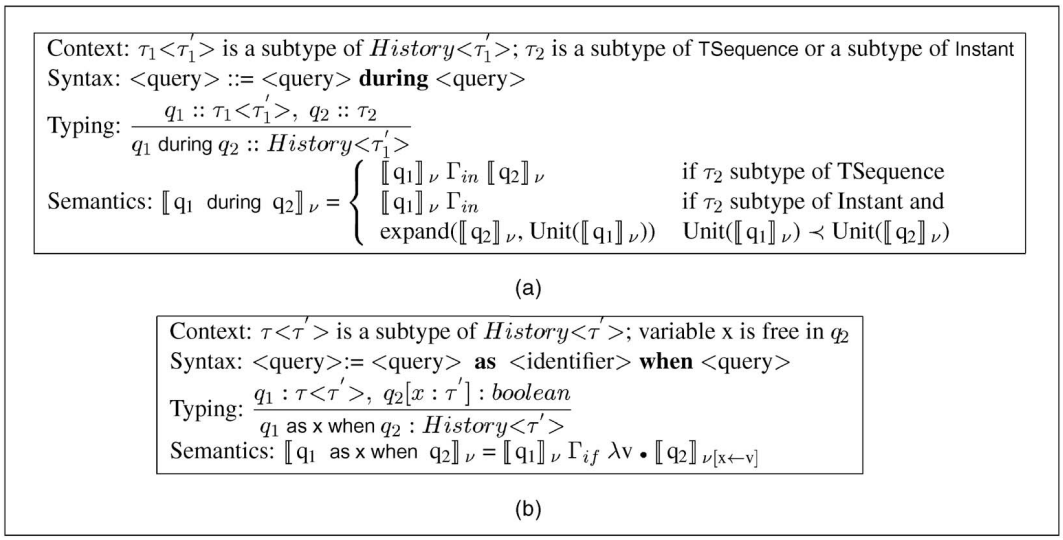

Fig. 7. Semantics of TEMPOQL's restriction constructs. (a) During: history restriction according to temporal values. (b) When: history restriction according to the value.

- A *syntax* given in a BNF-like notation with terminal symbols typeset in boldface.
- The *typing* rules for the construct using the notation $\frac{\text{premise}}{\text{implication}}$. The notation `q::t` means that the query `q` has type `t`, while `q[x::t']::t` means that query `q` has type `t` assuming that variable `x` has type `t'`.
- The *semantics* described in terms of expressions involving operators of the TEMPOS historical model. The semantics of a query is parameterised by a valuation function which determines the values of free symbols in the query. The notation $\nu[x \leftarrow v]$ denotes the valuation equal to ν except that it assigns value v to x. The preconditions that apply to the operators defining the semantics of a construct (in particular those related to the observation units of the argument histories), also apply to the construct itself.

As an example, the formalization of the restriction operators on histories is given in Fig. 7. For the formalization of the other TEMPOQL constructs, the reader is referred to the Appendix.

4.1.2 Temporal Restriction and Join

The restriction operators on histories (see Section 3.1.2) appear in TEMPOQL in the form of two language constructs, namely, `during` and `when`. The `during` construct generates a history by restricting a given history to those instants lying in a given set of instants. The `when` construct generates a history by restricting a given history to those instants when its value satisfies a given condition. The semantics of the `during` is defined in terms of the Γ_{in} operator (see Fig. 12a in the Appendix), while the semantics of the `when` is defined in terms of the Γ_{if} operator (see Fig. 12b). The following query illustrates these two constructs.

Q.1: Range and domain restriction. For each assembly line, retrieve its number and the set of instants when its production is greater than 100.

```
/* type: bag<struct<L: string,P:TSequence>> */
select struct(L: li.lineCode,
P: domain(li.production as p when p > 100))
from TheLines as li
```

The `domain` operator retrieves the set of instants at which the history given as parameter is defined. In the above example, this is the set of instants at which the history of the assembly line's production fulfils the condition given in the `when` clause. The `when` operator can also be used in conjunction with the `map` operator, as formalized in the Appendix.

The inner and outer temporal join operators on histories are also transposed to TEMPOQL in the form of two language constructs, namely, `join` and `ojoin`. The syntax and semantics of the `join` is formalized in the Appendix. The `ojoin` construct is defined in a similar way as the join, except that its semantics involves the outer join operator on histories instead of the inner join. The following query illustrates the use of the `ojoin`.

Q.2: Temporal join. When was the production of assembly line L1 greater than the production of assembly line L2?

```
/* type of result: TSequence */
element (select domain ( join
        (p1: L1.production, p2: L2.production)
         as j when j.p1 > j.p2) from TheLines as
         L1, TheLines as L2 where L1.lineCode =
         "L1" and L2.lineCode = "L2")
```

Together, the `when`, `join`, and `ojoin` constructs allow developers to express any query in which one or several histories are combined and restricted to those instants at which a given condition on their "simultaneous" values holds. The `map` operator can then be used to apply an arbitrary function to each of the values of the resulting history. These operators are inspired from those found in the temporal N1NF relational algebras of Clifford and Crocker [25] and Tansel [26] (among others).

As an alternative to expressing queries on simultaneous values through composition of the above constructs, TEMPOQL also integrates the *pointwise generalization principle*. In a nutshell, the pointwise generalization principle is a metaoperator that transparently combines a number of histories into a single one and pointwisely applies arbitrary functions (or operators) over the resulting combination, just as the composition of the `join` and the `map` would do. In this way, nontemporal operators are naturally given a temporal semantics, thereby achieving the S-reducibility requirement defined in [15].

4.1.3 Pointwise Generalization of OQL Constructs

For each OQL construct, TEMPOQL provides a counterpart construct on histories which seamlessly generalizes it. The semantics of the "extended" operator is defined according to the following generalization principle: given an N-ary operator θ: $\tau_1, \ldots \tau_n \rightarrow \tau_{n+1}$, an operator θ:

$$\text{History}\langle\tau_1\rangle, \ldots \text{History}\langle\tau_n\rangle \rightarrow \text{History}\langle\tau_{n+1}\rangle$$

is defined such that:

$$\forall \mathtt{i} \in \mathtt{Domain}(\mathtt{h}_1 \cap \ldots \cap \mathtt{h}_n)$$
$$\mathtt{Value}(\mathtt{h}_1\ \theta \ldots \theta\ \mathtt{h}_n, \mathtt{i}) = \mathtt{Value}(\mathtt{h}_1, \mathtt{i})\ \theta \ldots \theta\ \mathtt{Value}(\mathtt{h}_n, \mathtt{i}).$$

For example, given two queries `h1` and `h2` both retrieving histories of integers and given an arithmetic operator θ, query `h1` θ `h2` retrieves the history obtained by applying operator θ to synchronous values of `h1` and `h2` (see the Appendix). The same holds for comparison and Boolean operators.

Using the generalization principle, it is possible to reason about simultaneous values in a transparent way. Specifically, the developer of a query can write query expressions involving temporal properties, just as if these properties were not temporal. The generalization principle provides a semantics to these operators, whereby the nontemporal expression is applied to each "snapshot" of the involved properties, and the resulting values are joined into a single history. As an example, we show an alternative expression of query Q.2 that exploits the pointwise generalization principle.

Q.2: Pointwise generalization of arithmetic operators

```
element ( select domain ((L1.production >
        L2.production) as b when b) from
        TheLines as L1, TheLines as L2
        where  L1.lineCode = "T1"and
               L2.lineCode = "T2")
```

In this query, the expression `L1.production > L2.production` retrieves a history of Booleans. For a given instant, this history is true if the value of `L1.production` at this instant is greater than that of `L2.production`. The history of Booleans is then restricted to those instants when its value is "true," through the expression `h as b when b`.

OQL's navigation operator on structured types and objects is similarly generalized to deal with histories. If `h` is a query yielding a history whose values are objects with some attribute `a`, then query `h.a` is a history with the same temporal domain as `h`, obtained by projecting each value of `h` over attribute `a`. Formally, if `q` is a query retrieving a history of objects of class C, and `P` is a temporal property over class `C`, then:

$$[\![\mathtt{q.P}]\!]_\nu = \{\langle\, \mathtt{i}, \mathtt{o}'\,\rangle \mid \langle\, \mathtt{i}, \mathtt{o}\,\rangle \in [\![\mathtt{q}]\!]_\nu \wedge \langle\, \mathtt{i}, \mathtt{o}'\,\rangle \in \mathtt{o.P}\}.$$

The use of this "temporal navigation" operator is illustrated by the following query.

Q.3: Pointwise generalization of the navigation operator. When did the assembly line supervised by employee X have a quality-weighted production greater than that of the assembly line supervised by Y?

```
/* type of result: TSequence */
element(select domain
        (supX.supervises.production *
         supX.supervises.quality >
         supY.supervises.production *
         supY.supervises.quality as b when b)
         from  TheSupervisors as supX,
               TheSupervisors as supY
         where  supX.name = "X" and
                supY.name = "Y")
```

The reader can compare this query to the equivalent one in TOOBIS TOQL given in Section 2.3. In addition to the fact that the expression in TEMPOQL is considerably more concise, one can notice a difference in the level of abstraction between these two query expressions. Indeed, in TEMPOQL, the query expression captures a navigation from a `Supervisor` object to its associated assembly line and from there to the production volume and quality of this assembly line. In TOOBIS TOQL, this navigation only appears indirectly, in the form a succession of nested iterations and temporal join conditions.

The expression of the above query directly in OQL (without any temporal extension) is even more complex than that in TOOBIS TOQL since OQL does not provide operators on time intervals as TOQL does. It follows that the generalization principle in TEMPOQL adds considerable value with respect to a "pure OQL" approach.

The generalization principle applies to collection types as well: OQL collection expressions `forall`, `exists`, `sum`, `avg`, etc., are generalized in TEMPOQL to apply to histories of collections. For example, if `h` is a query retrieving the history of workers of an assembly line, then `count(h)` retrieves the history of the number of workers in that line.

Q.4: Pointwise generalization of operators on collections. When was the number of workers in line L1 greater than the number of workers in line L2?

```
/* type of result: TSequence*/
element(select domain(count(L1.workers) >
        count(L2.workers) as b when b)
        from TheLines as L1, TheLines as L2
        where L1.lineCode = "L1"
        and L2.lineCode = "L2")
```

The integration of the pointwise generalization principle distinguishes TEMPOQL from the temporal N1NF relational algebras of [25] and [26] (among others). On the other hand, there are clear connections between TEMPOQL's pointwise generalization principle and the "WHILE" clause of Gadia

and Nair's TempSQL [19].[3] The pointwise generalization principle has also been applied in [14], [15], where it appears in the form of a statement modifier called "sequenced," and, more recently, in [27], where it is applied to the design of operators over spatio-temporal datatypes. In fact, the idea behind this principle can be traced back to programming languages for (synchronous) stream processing such as LUSTRE [28].

4.1.4 Aggregations and Grouping

The `map ... on ... when` construct on histories is extended with a `group by` and a `having` clauses, accounting for history partitioning (see the Appendix). The partitioning criteria may be a temporal unit or a duration. Similar remarks to those formulated for the `map ... on ... when` construct apply, and the `when` and the `having` clauses are optional. Keyword `partition` is used in the `map` and `having` clauses to refer to the subhistories generated by the `group by` clause.

Q.5: Aggregations and duration-based grouping. For each assembly line and for each 10-day period during which the total production of this line is greater than 10,000, retrieve its average production on that period.

```
/* type: bag⟨struct⟨L: string, avgP :
History⟨real⟩⟩⟩ */
select struct(L: L.lineCode, avgP :
              map avg(partition)
              on L.production as p
              group by #"10 days"
              having sum(partition)
              > 10000)
from TheLines as L
/* The result of the subquery introduced
by the map clause, is a history at the
granularity of the day: for a given day,
the average production of the 10-day period
starting on that day is retrieved, provided that
the total production on that period is
> 10000 */
```

Another query illustrating the `map ... on ... when` construct is given Section 5.3.

4.1.5 Reasoning about Succession in Time

The `afterfirst`, `beforefirst`, `afterlast`, and `beforelast` constructs are straightforward adaptations of the corresponding operators on histories (see the Appendix). Their syntax is similar to that of the `when` construct.

Q.6: Succession in time-splitting histories. For each line, retrieve its production history since the last time that this production was smaller than 100.

```
/* type: set⟨L: string, P:
History⟨unsigned long⟩⟩ */
select struct (L: L.lineCode, P: L.production
              as b afterlast b ≥ 100)
from TheLines as L
```

3. In fact, [19] considers queries similar to Q.3 above.

By composing the four splitting operators, it is possible, in theory, to express queries involving complex sequences of events. However, the complexity of these query expressions rapidly increases with the complexity of the involved sequence, making this an impractical approach. In addition, when duration constraints are involved, these constructs must be composed with the "group by" construct previously discussed, leading to still more complicated query expressions. To tackle this complexity, we defined a language for expressing patterns of histories (see [29] for details), and integrated this language in TEMPOQL through an operator called `matches`. This operator takes as its parameter a history and a pattern and returns a Boolean stating whether the pattern occurs in the history. The language for describing patterns is based on timed regular expressions: It provides operators such as `followed by` (to express sequencing) `at most` and `at least` (to express repetition with or without temporal constraints). As a result, the operator `matches` can be evaluated using efficient automata-based techniques. A flavor of this language is given in Section 5.3, Query Q.8.

The above operators for reasoning about succession in time are a distinguishing feature of TEMPOQL. In related proposals such as TOQL [7], TempSQL [19], and the temporal N1NF relational algebras of [25] and [26], the only way to express queries about succession in time is by performing cartesian products and explicitely manipulating the timestamps of the resulting tuples.

4.2 Pointwise Temporal Object Browsing

The pointwise browser is designed to address time-related users' tasks such as:

- Analyze data about the supervisor and workers of each assembly line at a given date.
- Compare at different dates, a given worker's salary with respect to that of the supervisor of the assembly line to which he is assigned.
- Find out whether the composition of a given assembly line (equipment plus workers) considerably changes when its supervisor does.

4.2.1 Overview

The pointwise browser interface (see Fig. 8) is made up of two parts: a *time line window* and a tree of *snapshot windows*. A snapshot window displays either a nontemporal object or a snapshot of a temporal object at a given instant. The instant with respect to which the object snapshots are determined is the same for all the windows in the tree and is subsequently called the *reference instant*. The reference instant is constrained to reside within a given interval called the *temporal browsing range*.

The role of the time line window is to set the reference instant. At the beginning of a session, the reference instant is at the middle of the temporal browsing range. Its position varies thereafter according to the user interactions with the sliders and buttons composing the time line window. In its simplest form, the time line window is composed of a slider (called the *main slider*) and four buttons placed at the ends of this slider. Two of the buttons (labeled with simple arrows) allow the user to move the reference instant

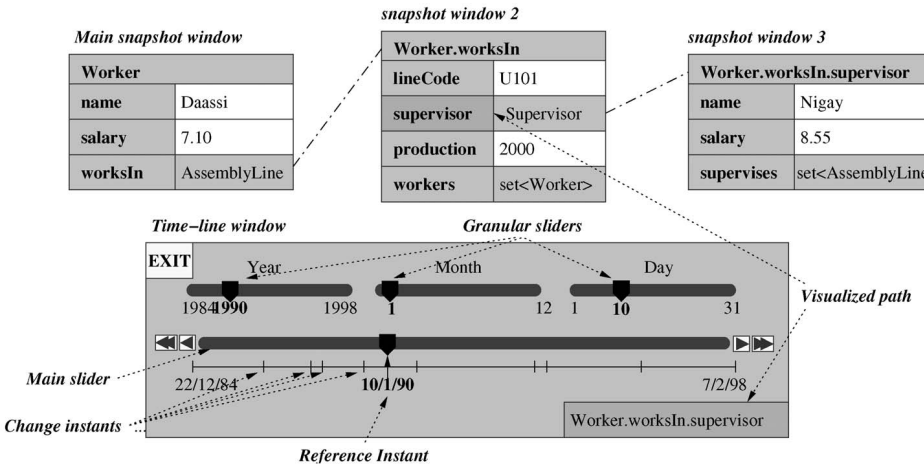

Fig. 8. The pointwise temporal object browser. The schema of the underlying database is the one given in Section 2.1. In particular, properties `Worker::salary` and `Worker::worksIn` are temporal.

forward or backward by one unit. The other pair of buttons (labeled with double-arrows) allow to move the reference instant to the next/previous instant where the value of a given navigation path (called the *visualized path*) changes. The instants at which the value of the visualized path changes are called *change instants*. Change instants are visually represented as vertical marks lying within a horizontal line just beneath the main slider.

In Fig. 8, the change instants are those when the supervisor of worker "Daassi" changes, whether this change is due to the fact that this worker is assigned to a new assembly line and that this assembly line has a different supervisor than the previous one, or to the fact that the supervisor of the assembly line to which this worker is assigned changes.

In addition to the main slider, the time line window may additionally contain several *granular sliders*, which allow the user to move the reference instant with different "steps" according to a given calendar. For instance, if the reference instant is a date and that the user specifies the calendar Year/Month/Day (as in Fig. 8), three granular sliders appear in the timeline window: The first one allows one to move the reference instant with a step of a year, the second one with a step of a month, and the third one with a step of a day (within the limits of a given month).

Snapshot windows are structured as forms containing one line per property of the visualized object or object snapshot. Each line is composed of two buttons: the left one labeled with the name of the property and the right one labeled with its value at the reference instant.[4] The value of a nontemporal property is always the same regardless of the reference instant. The value of a temporal property at a given instant is equal to the value of its history at that instant, which is itself defined as follows:

- The value at instant `I` of a history represented as an instant time-stamped collection of objects is equal to the object in this collection whose time stamp is equal to `I`. If no such object exists, the value is null.
- The value at instant `I` of a history represented as an interval time-stamped collection of objects is equal to the object in this collection whose timestamp contains `I`. If no such object exists, the value is null.

All buttons within a snapshot window are clickable, except those which denote literal values (i.e., integers, reals, string, and characters). For instance, in Fig. 8 all the buttons within the snapshot windows are clickable, except the white-colored ones.

At the beginning of a session, there is a single snapshot window. Other snapshot windows are incrementally added according to the user interactions with the clickable buttons denoting object references which appear within existing forms. The object displayed by a given snapshot window other than the main one is equal to the object referenced by the button from which this window was opened. For instance, the configuration shown in Fig. 8 is obtained by displaying the worker named "Daassi" and successively clicking on the buttons labeled `AssemblyLine` and `Supervisor`.

The user may also click on the buttons labeled with property names (i.e., the buttons on the left column of a form). The semantics of this interaction is that the selected property becomes the visualized path expression and the set of "change instants" attached to the timeline window are updated accordingly. For instance, clicking on the button labeled `production` on window 2 of Fig. 8 sets the visualized path to be `Worker.worksIn.production` instead of `Worker.worksIn.supervisor`. The vertical marks drawn on the line just below the main slider, and the label appearing in the lower right corner of the timeline window are then modified accordingly.

Whenever the user modifies the reference instant, the new reference instant is notified to the main snapshot window (see Fig. 9). Upon receiving this notification, the main window computes the snapshot at the new reference instant of the object that it displays and updates its appearance so as to reflect this new snapshot. During this process, if the value of a temporal property changes, the new value is transmitted to its dependent window if any. Finally, the main window propagates the notification of the

4. If the value of a property is not printable (i.e., its type is not integer, string, etc.), the name of its class is used as its label.

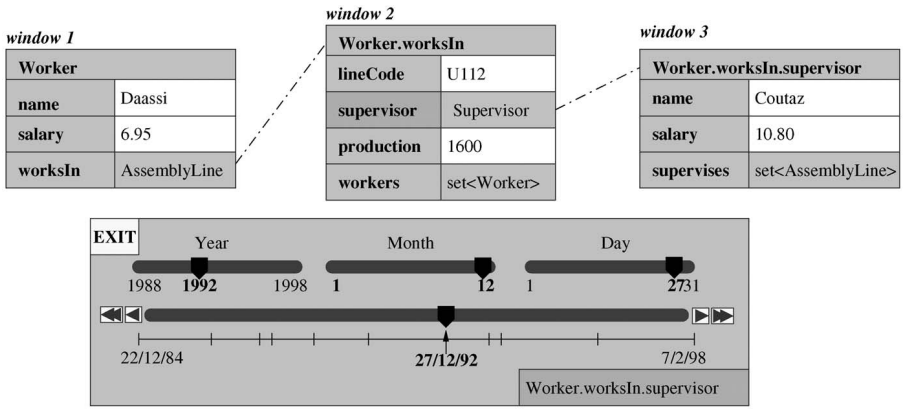

Fig. 9. Modification of the reference instant upon the configuration given in Fig. 8.

new reference instant to all its dependent windows, and the above process is carried out recursively.

As stated before, the reference instant is constrained to lie within an interval called the temporal browsing range. This range is taken to be the smallest interval containing all the instants when at least one of the temporal properties of the object displayed by the main snapshot window (also called the *main object*) is defined. For example, if the main object is a worker `W`, such that:

```
W.salary = set(struct(timestamp: [1..4],
                      value: 10.0),
               struct(timestamp: [6..9],
                      value: 12.0))
```

and

```
W.worksIn = set(struct(timestamp: [2..4],
                       value: X),
                struct(timestamp: [6..8],
                       value: Y)),
```

(where X and Y are two assembly lines) then the temporal browsing range is taken to be interval [1..9].

4.2.2 Pointwise Browsing in the Presence of Null-Valued Properties

Heretofore, we have implicitly assumed that all properties displayed within snapshot windows have nonnull values. However, null-valued properties within a snapshot window may arise in two cases:

- The value of the property at the reference instant was actually set to "null" through an update (this can occur whether the property is temporal or not).
- The history of a temporal property is not defined at the reference instant in which case its value is null. This situation can occur in the middle of a pointwise browsing session since the temporal browsing range may include instants in which some of the temporal properties of the main object are defined while others are not.

We visually denote a null value through a filled rectangle. However, this does not solve all the problems arising from nulls. Indeed, suppose that the worker displayed in Fig. 9 is not assigned to any assembly line on 1/5/97 (i.e., there is no element in its history whose timestamp contains this date). If the reference instant is set to this date, the value of property `worksIn` becomes null, and something has to be done with its dependent forms (i.e., windows 2 and 3 in Fig. 9).

In the pointwise visual browser, if following a modification of the reference instant, one of the properties displayed by a snapshot window becomes null and, if this property has a snapshot window attached to it, then all the windows in the subtree stemming from this property become *inactive*. Inactivity of a snapshot window is rendered by modifying the appearance of some elements within the window, e.g., graying out the labels denoting property names and erasing the labels denoting property values. Labels are restored to their normal appearance when the window becomes active again.

4.2.3 Pointwisely Browsing Collections of Temporal Objects

To accommodate collections, we augment the pointwise browser with the concept of *synchronous navigation* as defined in object browsers such as PESTO [30]. Basically, a collection of temporal objects is displayed in the same way as a single one, except that the corresponding snapshot window contains a couple of arrow-labeled buttons on top of it. This window displays the snapshot at the reference instant of one of the objects within the collection. Clicking on either of the arrows allows one to switch to the next or the previous object in the collection (see Figs. 10a and 10b).

As before, the temporal browsing range is defined with respect to the object visualized by the main snapshot window. Therefore, when this object changes, the browsing range is recomputed. This is the reason why the timeline is redrawn when transitioning from configuration 1 (see Fig. 10a) to configuration 2 (see Fig. 10b). During this transition, the change instants are also recomputed. Under some circumstances this computation involves a relatively large amount of data. For instance, consider the example of Figs. 10a and 10b and suppose that the visualized path expression is `Workers.worksIn.supervisor.salary` (meaning that the path `Workers.worksIn.supervisor` is displayed) computing the change instants involves the following histories:

- The history of the employee's assembly lines.

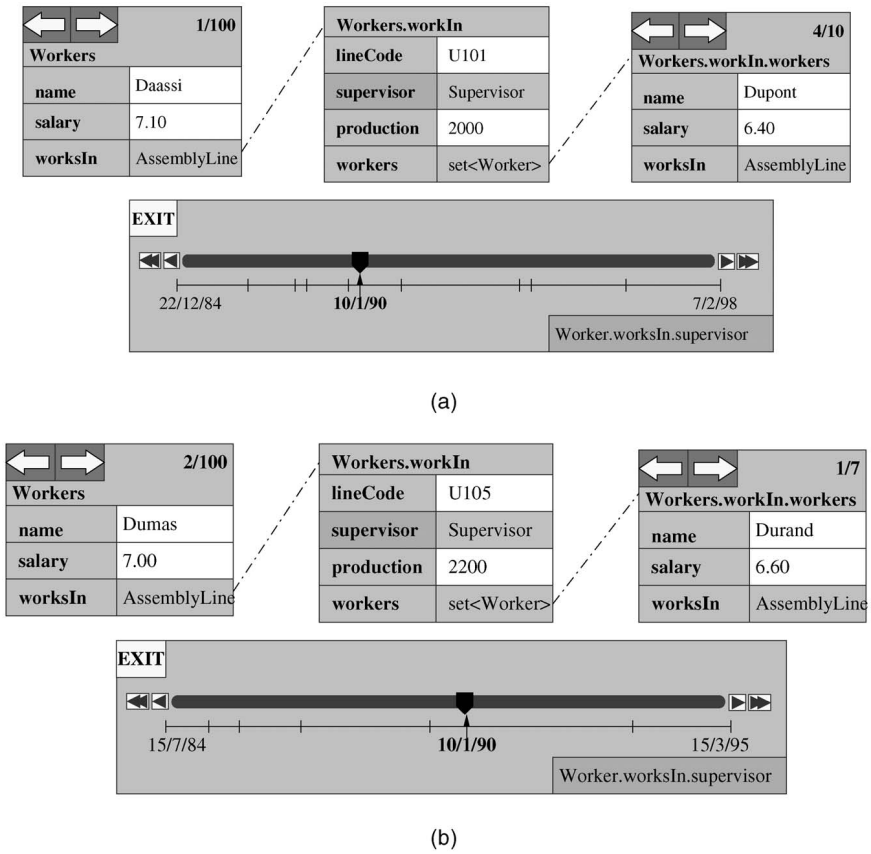

Fig. 10. Pointwisely browsing a collection of temporal objects. Configuration 2 is the result of clicking on the right arrow of the main snapshot window in configuration 1.

- The history of the supervisors of each line where the visualized employee has ever worked.
- The history of the salary of each supervisor appearing in any of the histories referenced in the previous item.

4.2.4 Comparison with Related Work

Data visualization has received little attention within the temporal database literature. Instead, many efforts have focused on the design of visual query languages for temporal databases (see, for example, [31]). Such languages are clearly complementary to data visualization techniques.

In the area of information visualization, many techniques for graphically displaying and browsing temporal data have been designed [32]. Most of these techniques are oriented toward quantitative time series, i.e., periodical series of numerical data items. Plaisant et al. [33] adapts some concepts developed in this area to design an interface for visualizing legal and medical records involving non-quantitative temporal data.

The pointwise temporal object browser is an extension of data browsers such as O_2Look [34], ODEVIEW [35], and PESTO [30]. Nevertheless, the pointwise browser considerably differs from all the above ones since it treats time as a dimension per se.

5 Implementation and Applications

5.1 Overall Architecture

TEMPOS has been implemented on top of the object-oriented DBMS O_2. Fig. 11 depicts the prototype architecture. It essentially consists of a library of classes corresponding to the ADT hierarchies defined in the time and historical models, two preprocessors implementing TempODL and TEMPOQL, respectively, and a visualization module. A temporal metadata manager handles the communication between the preprocessors.

The library of time-related classes has been primarily implemented in the O_2's database programming language O_2C (which is translated into C by a preprocessor provided by the O_2 system), while the preprocessors and the metadata manager have been implemented in C using tools such as Lex and Yacc. These classes can be used from C++ and Java programs using the corresponding bindings provided by the O_2 system.

The visualization module implements the pointwise object browser and some other visualization techniques that we have developed as part of a work on temporal visual data analysis. As the preprocessor, this module uses metadata related to the temporal classes and properties defined in the database's schema.

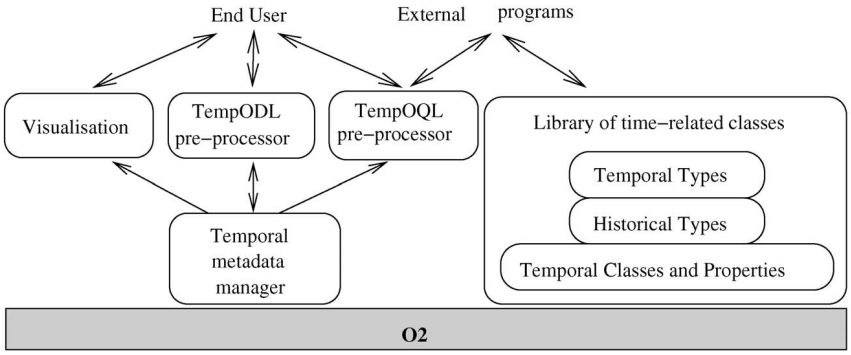

Fig. 11. Prototype architecture.

5.2 Implementation Issues Related to the Lack of Parametric Classes in ODMG

The major problems that we faced during the design and implementation of the TEMPOS prototype, were those related to the lack of parametric classes in the O_2 model (which is true of the ODMG object model as well). Indeed, the `History` datatype could be naturally mapped into a parametric class.

One of the solutions that we envisaged, is to generate a class for each kind of history involved in an application (e.g., one for histories of integers, another for histories of floats, etc.). However, in realistic situations, this rapidly leads to a high proliferation of classes. In addition, some operators, such as the temporal joins, cannot be satisfactorily implemented using this approach since the type of the values of the resulting history intrinsically depends on that of the argument histories (see Section 3.1.2).

Instead, we decided to partially simulate parametric classes by exploiting the preprocessors included in the architecture. In this approach, a single nonparametric class `History`, corresponding to histories whose values are of type `Object` (the top of the ODMG's class hierarchy), is first implemented. During schema definition, each history-valued attribute is declared as being of type `History` by the TEMPODL preprocessor, but its exact type specification is stored in the temporal metadata manager. By accessing this metadata manager, the TEMPOQL preprocessor gets to know the exact type of the histories involved in a query. With this knowledge, the preprocessor adds explicit downcastings in the translated query expression whenever the value of a history is involved. In this way, the user of TEMPOQL manipulates histories as if they were parametrically typed.

5.3 Applications

Two applications have been implemented using TEMPOS. In the first one, which is detailed in [36], we applied the concepts of timeline, granularity, and history to model the structure and the annotations of video documents. This application highlighted the close connections between the concept of granularity and histories in temporal databases and the concepts of structuration and annotations in video databases.

The second application, which we overview in this section, dealt with the analysis of the use of resources and space over time at the ski resort "Valloire" located in the French Alps [37]. The development of this application was part of a broader study on the use of the resort's infrastructures, aiming at formulating proposals for introducing new activities and improving the use of the environment and the existing infrastructures. A survey on 200 tourists and inhabitants was conducted in the ski resort. In this survey, people described their activities (a history of activities), companionships (a history of companionships), and movements (a history of locations) during the day(s) preceding the survey. After digitalizing this information, no adequate tool was found to analyze this data due to the tight connections between their spatial and temporal components.

TEMPOS was used to model and store data produced by the survey and TEMPOQL was used to query the resulting histories. This experience highlighted that the operators on histories provided in TEMPOS, matched the needs of the studied application. In particular, operators such as the history restriction (`during` and `when`) and the history join (`join`) allowed us to deal with queries related to the context under which individuals performed their activities. On the other hand, the operators for grouping and aggregating allowed us to deal with queries related to the identification of routines as illustrated below.

Q.7: Change of granularity. For each person and for each day, how long did this person skied?

```
/* type of result: bag⟨struct⟨p: Person,
skied: History⟨Duration⟩⟩⟩ */
select struct (person : p, skied :
       map duration (partition) on p.activity
       when a.name = "ski" group by Day)
from ThePersons as p
/* Variable 'partition' denotes the history of
skiing periods during a day. Operator
'duration' retrieves the size of the domain of a
history. */
```

The operators for reasoning about succession in time revealed to be useful when analyzing chains of activities performed by individuals in the resort. For example, the

following query illustrates the use of the operator `matches` mentioned in Section 4.1.5.

Q.8: Succession in time. Which persons went skiing just after shopping and took dinner at most three hours after skiing?

```
select p from ThePersons as p where p.activity
 as a matches a.name = "shopping" followed by
 a.name = "skiing" followed by at most ♯"3 hours"
 followed by a.name = "dinner"
/* A person is selected if, at least one time,
she(he) went shopping and then when skiing, and
then did anything during at most three hours and
after, took dinner. */
```

Our experience with the "Valloire" application put forward the need to introduce in TEMPOS other semantic assumptions than the three discussed in Section 3.2 and, especially, semantic assumption dealing with spatially-valued temporal properties. Indeed, the movements of a person (a history of locations) is a temporal attribute with a spatial structural domain type (a location has a geometry associated to it). The semantic assumption of this temporal attribute was modeled as stepwise, which means that, for a given object, the value of this property between two successive instants in the effective history remains constant. It is straightforward to see that this cannot accurately model continuous and arbitrary movement. Instead, a movement over some interval of time is modeled under this approach by associating a segment or polyline denoting a path to each instant in this interval. In the setting of the Valloire application, this fitted well the requirements initially expressed since, in the surveys, people represented their movements by polylines whose segments were annotated by time intervals, and that no assumption was made a priori on the kind of movement people had. It became clear at the end of our experience that the application data and queries should be revisited under other semantic assumptions. For example, under some conditions, it would be natural to assume that individuals move at constant speed between the two edges of a path. Alternatively, this assumption could be replaced with a more complex one, which takes into account the fact that people in a car move slower in downtown than in a road. Ongoing research in spatio-temporal data models and languages [27] is addressing these interpolation aspects.

6 CONCLUSION

TEMPOS is a comprehensive temporal database framework which synthesizes and unifies many concepts, requirements, and functionalities recognized as necessary to extend existing DBMS for managing temporal data. This framework includes a history model, a temporal object model, a query language, and a visual browser.

The history model provides an abstract datatype dedicated to the notion of history and a rich set of representation independent operators over it. This set of operators includes temporal extensions of the selection, projection, and join operators of the relational algebra, operators for partitioning (grouping) according to a granularity or a duration, and algebraic versions of temporal logic operators. These operators, together with those defined by the time model, form the basis for TEMPOQL: the proposed extension of OQL. Of course, a developer can add other operators by specializing the interfaces and classes of the historical model.

The temporal object model, which extends the ODMG model, fulfills two requirements related to legacy code migration: upward compatibility and temporal transitioning support. These requirements have been defined in the context of relational models in [15]. The first requirement states that a database may be transparently migrated from an ODMG system to a temporal extension of it. The second requirement enables nontemporal legacy code to remain operational even after a database schema is modified to add temporal support to some of its components. In TEMPOS, temporal transitioning support is ensured by separating temporal properties from the history of their values: A temporal property has a historical value in the context of a temporal application and a "snapshot" value in the context of a nontemporal one. Update operators on temporal properties are defined in such a way that updates done by nontemporal applications are compatible with those performed by temporal ones. The concept of "now," which has lead to many confusions in previous temporal data models [38], is modeled by dynamically generating a history from a now-relative temporal property.

The TEMPOQL query language offers facilities to express, in a unified framework, classical temporal queries such as restriction, join, and grouping, together with operators for reasoning about succession in time. With respect to related proposals, the main originalities of TEMPOQL are:

- It is representation-independent in the sense that histories are primarily manipulated through operators whose semantics is not tight to a particular representation. This contrasts with previous proposals such as [16], [17], [7] and [8], where queries on histories are expressed by applying iterators on collections of interval time-stamped values representing them.
- It integrates two algebraic history grouping operators. While these operators are present in other temporal query languages (e.g., TSQL2 [5]), they are usually not defined algebraically. The algebraic nature of these operators in TEMPOQL facilitates their composition with the other operators.
- By applying the pointwise generalization principle, TEMPOQL extends the semantics of standard OQL constructs to handle histories. The integration of the pointwise generalization principle enables TEMPOQL to fulfill the S-reducibility requirement defined in [15].

Regarding data visualization, TEMPOS integrates a novel technique for browsing collections of temporal objects. This technique orthogonally supports three kinds of navigation:

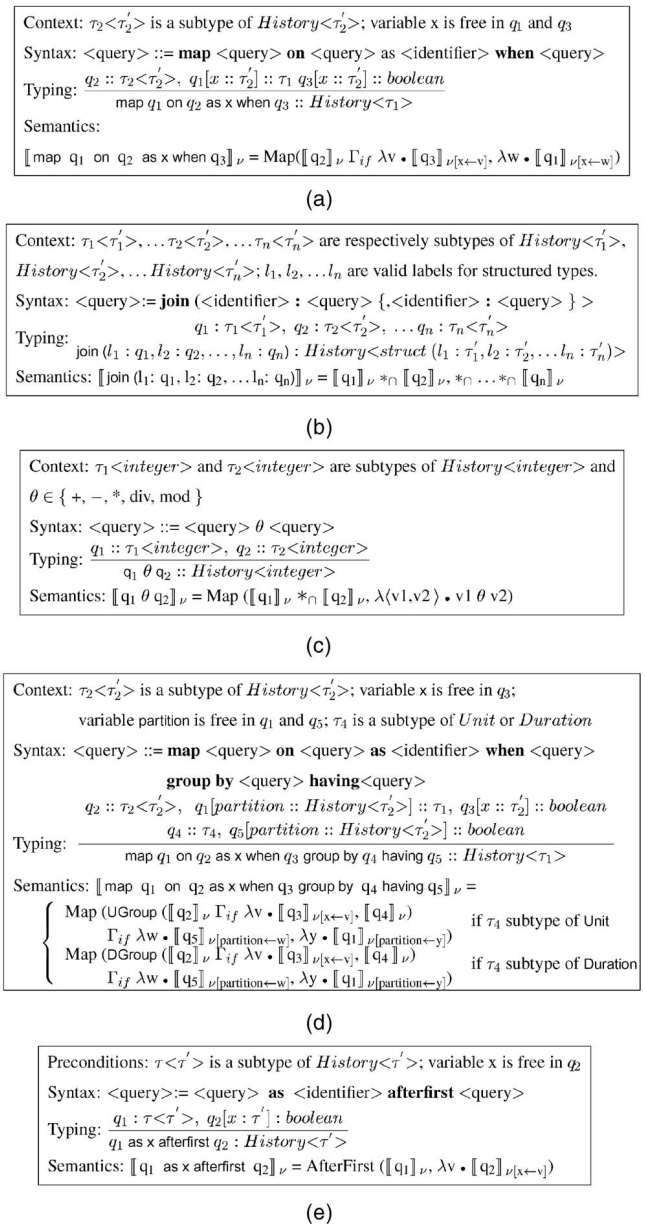

Fig. 12. (a) Map: history projection. (b) Join: merging histories. (c) Generalization of the arithmetic operators on integers to two historical arguments. (d) Group by: temporal grouping. (e) Afterfirst: succession in time-splitting histories.

1) navigation through time, 2) navigation via object relationships, and 3) navigation within the elements of a collection.

All the above models and languages have been formalized at the syntactical and the semantical level, and a prototype on top the O_2 DBMS has been developed on the basis of this formalization. This prototype has been used to develop several applications from various contexts (GIS, time series, and multimedia).

As a future work, we envisage two main research avenues: 1) designing user interfaces for visually mining temporal and spatio-temporal data and 2) applying the TEMPOS framework to model and query traces of inter-organizational business process executions.

APPENDIX

FORMALIZATION OF TEMPOQL

The definitions presented in Fig. 12, together with Fig. 7, formally describe the extensions that TEMPOQL brings on top of OQL. The formalization style and the notations used are discussed in Section 4.1.1.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers of the *IEEE Transactions on Knowledge and Data Engineering* for their valuable feedback.

REFERENCES

[1] *Temporal Databases,* A.U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass, eds., The Benjamins/Cummings Publishing, 1993.
[2] R.T. Snodgrass, "Temporal Object-Oriented Databases: A Critical Comparison," *Modern Database Systems. The Object Model, Interoperability and Beyond,* W. Kim, ed., chapter 19, Addison Wesley, 1995.
[3] *Temporal Databases: Research and Practice.* O. Etzion, S. Jajodia, and S.M. Sripada, eds., Springer Verlag, 1998.
[4] R.T. Snodgrass, *Developing Time-Oriented Database Applications in SQL.* Morgan Kaufmann, July 1999.
[5] *The TSQL2 Temporal Query Language.* R.T. Snodgrass, ed. , Kluwer Academic, 1995.
[6] *The Object Database Standard: ODMG 3.0.* R.G.G. Cattell and D. Barry, eds., Morgan Kaufmann, Jan. 2000.
[7] A. Sotiropoulou, M. Souillard, and C. Vassilakis, "Temporal Extension to ODMG," *Proc. Workshop Issues and Applications of Database Technology (IADT),* July 1998.
[8] E. Bertino, E. Ferrari, G. Guerrini, and I. Merlo, "Extending the ODMG Object Model with Time," *Proc. European Conf. Object-Oriented Programming (ECOOP),* July 1998.
[9] J.-F. Canavaggio, "TEMPOS, un Modèle d'Historiques pour un SGBD Temporel," Thèse de doctorat, Universitè Joseph Fourier, Grenoble, France, Nov. 1997.
[10] M.-C. Fauvet, M. Dumas, and P.-C. Scholl, "A Representation Independent Temporal Extension of ODMG's Object Query Language," *Actes des 15e Journées Bases de Données Avancées,* Oct. 1999.
[11] M. Dumas, M.-C. Fauvet, and P.-C. Scholl, "Updates and Application Migration Support in an ODMG Temporal Extension," *Proc. First Int'l Workshop Evolution and Change in Data Management,* Nov. 1999.
[12] M. Dumas, C. Daassi, M.C. Fauvet, and L. Nigay, "Pointwise Temporal Object Database Browsing," *Proc. ECOOP Symp. Objects and Databases,* June 2000.
[13] "The Consensus Glossary of Temporal Database Concepts—February 1998 Version,"C.-S. Jensen and C.-E. Dyreson, eds., Springer, 1998.
[14] R.T. Snodgrass, M. Bohlen, C. Jensen, and A. Steiner, "Transitioning Temporal Support in TSQL2 to SQL3," *Temporal Databases: Research and Practice,* 1998.
[15] M. Bohlen, C.S. Jensen, and R.T. Snodgrass, "Temporal Statement Modifiers," *ACM Trans. Database Systems,* vol. 25, no. 4, pp. 407-456, Dec. 2000.
[16] I.A. Goralwalla and M.T. Ozsu, "Temporal Extensions to a Uniform Behavioral Object Model," *Proc. 12th Int'l Conf. Entity-Relationship Approach-ER'93,* 1993.
[17] E. Rose and A. Segev, "TOOSQL-A Temporal Object-Oriented Query Language," *Proc. 12th Int'l Conf. Entity-Relationship Approach-ER'93,* 1993.
[18] D. Toman, "Point-Based Temporal Extensions of SQL and Their Efficient Implementation," *Temporal Databases: Research and Practice,* 1998.
[19] S.K. Gadia and S.S. Nair, "Temporal Databases: A Prelude to Parametric Data," *Temporal Databases,* 1993.
[20] X.S. Wang, S. Jajodia, and V.S. Subrahmanian, "Temporal Modules: An Approach Toward Federated Temporal Databases," *Information Systems,* vol. 82, 1995.

[21] M. Dumas, M.-C. Fauvet, and P.-C. Scholl, "TEMPOS: A Temporal Database Model Seamlessly Extending ODMG," Research Report 1013-I-LSR-7, LSR-IMAG, Grenoble, France, Mar. 1999.
[22] E.A. Emerson, "Temporal and Modal Logic," *Handbook of Theoretical Computer Science*, J. van Leeuwen, ed., vol. 2, 1990.
[23] R.T. Snodgrass and I. Ahn, "A Taxonomy of Time in Databases," *Proc. ACM SIGMOD Conf.*, May 1985.
[24] H. Riedel and M.H. Scholl, "A Formalization of ODMG Queries," *Proc. Seventh Int'l Conf. Data Semantics (DS-7)*, Oct. 1997.
[25] J. Clifford and A. Croker, "The Historical Relational Data Model (HRDM) Revisited," *Temporal Databases*, 1993.
[26] A.U. Tansel, "Temporal Relational Data Model," *IEEE Trans. Knowledge and Data Eng.*, vol. 9, no. 3, pp. 464-479, 1997.
[27] R. Guting, M. Bohlen, M. Erwig, C. Jensen, N. Lorentzos, M. Schneider, and M. Vazirgianis, "A Foundation for Representing and Querying Moving Objects," *ACM Trans. Database Systems*, vol. 25, no. 1, pp. 1-42, Mar. 2000.
[28] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A Declarative Language for Programming Synchronous Systems," *Proc. 14th ACM Symp. Principles of Programming Languages*, 1987.
[29] M. Dumas, M.-C. Fauvet, and P.-C. Scholl, "Handling Temporal Grouping and Pattern-Matching Queries in a Temporal Object Model," *Proc. CIKM Int'l Conf.*, Nov. 1998.
[30] M. Carey, L. Haas, V. Maganty, and J. Williams, "PESTO: An Integrated Query/Browser for Object Databases," *Proc. Int'l Conf. Very Large Databases (VLDB)*, Aug. 1996.
[31] S. Fernandes, U. Schiel, and T. Catarci, "Visual Query Operators for Temporal Databases," *Proc. Fourth Int'l Workshop Temporal Representation and Reasoning (TIME)*, May 1997.
[32] E. Tufte, *The Visual Display of Quantitative Information.* Graphics Press, 1984.
[33] C. Plaisant, B. Milash, A. Rose, S. Widoff, and B. Schneiderman, "LifeLines: Visualizing Personal Histories," *Proc. ACM CHI Conf.*, Apr. 1996.
[34] D. Plateau, P. Borras, D. Leveque, J.C. Mamou, and D. Tallot, "Building User Interfaces with Looks," *The Story of O_2*, F. Bancilhon, C. Delobel, and P. Kanellakis, eds., 1992.
[35] S. Dar, N.H. Gehani, H.V. Jagadish, and J. Srinivasan, "Queries in an Object-Oriented Graphical Interface," *J. Visual Languages and Computing*, vol. 6, no. 1, pp. 27-52, 1995.
[36] M. Dumas, R. Lozano, M.-C. Fauvet, H. Martin, and P.-C. Scholl, "A Sequence-Based Object-Oriented Model for Video Databases," *Multimedia Tools and Applications*, vol. 18, no. 3, Dec. 2002.
[37] M.-C. Fauvet, S. Chardonnel, M. Dumas, P.-C. Scholl, and P. Dumolard, "Applying Temporal Databases to Geographical Data Analysis," *Proc. DEXA Workshop Spatio-Temporal Data Models and Languages*, Sept. 1999.
[38] J. Clifford, C. Dyreson, T. Isakowitz, C. Jensen, and R. Snodgrass, "On the Semantics of "Now" in Databases," *ACM Trans. Database Systems*, vol. 22, no. 2, pp. 171-214, June 1997.

Marlon Dumas received the PhD degree from the University of Grenoble, France, in 2000, for his work on temporal databases. He is currently a lecturer at the Queensland University of Technology in Brisbane, Australia. His research interests include temporal data management, infrastructure for Web-based information systems, and E-commerce technologies. He is member of the IEEE Computer Society.

Marie-Christine Fauvet received the PhD degree in computer sciences from the University of Grenoble in 1988, for her work on version management for CAD Applications. Since 1988, she has successively been an assistant and associate professor at the University of Grenoble. Her current research interests include database schema evolution, object databases, temporal databases, databases for the Web, and Web services.

Pierre-Claude Scholl received the PhD degree in computer sciences from the University of Grenoble in 1979, for his works on programming methodology. He is currently a full professor at the University of Grenoble, France. His research interests include temporal databases and object-oriented models and languages.